# NumPy Reference

## *Release 1.19.0*

**Written by the NumPy community**

**June 29, 2020**

# CONTENTS

**1 Array objects** **3**

1.1 The N-dimensional array (`ndarray`) . . . 3
1.2 Scalars . . . 56
1.3 Data type objects (`dtype`) . . . 73
1.4 Indexing . . . 91
1.5 Iterating Over Arrays . . . 99
1.6 Standard array subclasses . . . 112
1.7 Masked arrays . . . 237
1.8 The Array Interface . . . 419
1.9 Datetimes and Timedeltas . . . 424

**2 Constants** **431**

**3 Universal functions (`ufunc`)** **439**

3.1 Broadcasting . . . 439
3.2 Output type determination . . . 440
3.3 Use of internal buffers . . . 440
3.4 Error handling . . . 441
3.5 Casting Rules . . . 444
3.6 Overriding Ufunc behavior . . . 445
3.7 `ufunc` . . . 446
3.8 Available ufuncs . . . 460

**4 Routines** **465**

4.1 Array creation routines . . . 465
4.2 Array manipulation routines . . . 513
4.3 Binary operations . . . 564
4.4 String operations . . . 575
4.5 C-Types Foreign Function Interface (`numpy.ctypeslib`) . . . 629
4.6 Datetime Support Functions . . . 632
4.7 Data type routines . . . 640
4.8 Optionally Scipy-accelerated routines (`numpy.dual`) . . . 659
4.9 Mathematical functions with automatic domain (`numpy.emath`) . . . 660
4.10 Floating point error handling . . . 660
4.11 Discrete Fourier Transform (`numpy.fft`) . . . 665
4.12 Financial functions . . . 693
4.13 Functional programming . . . 705
4.14 NumPy-specific help functions . . . 713
4.15 Indexing routines . . . 716
4.16 Input and output . . . 761

4.17 Linear algebra (`numpy.linalg`) . . . 793
4.18 Logic functions . . . 846
4.19 Mathematical functions . . . 873
4.20 Matrix library (`numpy.matlib`) . . . 985
4.21 Miscellaneous routines . . . 992
4.22 Padding Arrays . . . 1001
4.23 Polynomials . . . 1005
4.24 Random sampling (`numpy.random`) . . . 1224
4.25 Set routines . . . 1478
4.26 Sorting, searching, and counting . . . 1484
4.27 Statistics . . . 1504
4.28 Test Support (`numpy.testing`) . . . 1554
4.29 Window functions . . . 1579

**5 Global State 1593**
5.1 Performance-Related Options . . . 1593
5.2 Interoperability-Related Options . . . 1593
5.3 Debugging-Related Options . . . 1594

**6 Packaging (`numpy.distutils`) 1595**
6.1 Modules in `numpy.distutils` . . . 1595
6.2 Configuration class . . . 1601
6.3 Building Installable C libraries . . . 1610
6.4 Conversion of `.src` files . . . 1612

**7 NumPy Distutils - Users Guide 1613**
7.1 SciPy structure . . . 1613
7.2 Requirements for SciPy packages . . . 1613
7.3 The `setup.py` file . . . 1614
7.4 The `__init__.py` file . . . 1621
7.5 Extra features in NumPy Distutils . . . 1621

**8 NumPy C-API 1623**
8.1 Python Types and C-Structures . . . 1623
8.2 System configuration . . . 1640
8.3 Data Type API . . . 1642
8.4 Array API . . . 1647
8.5 Array Iterator API . . . 1689
8.6 UFunc API . . . 1706
8.7 Generalized Universal Function API . . . 1712
8.8 NumPy core libraries . . . 1715
8.9 C API Deprecations . . . 1721

**9 NumPy internals 1723**
9.1 NumPy C Code Explanations . . . 1723
9.2 Memory Alignment . . . 1730
9.3 Internal organization of numpy arrays . . . 1732
9.4 Multidimensional Array Indexing Order Issues . . . 1732

**10 NumPy and SWIG 1735**
10.1 Testing the numpy.i Typemaps . . . 1751

**11 Acknowledgements 1755**

**Bibliography 1757**

**Python Module Index** **1771**

**Index** **1773**

**Release**

1.19

**Date**

June 29, 2020

This reference manual details functions, modules, and objects included in NumPy, describing what they are and what they do. For learning how to use NumPy, see the complete documentation.

CHAPTER

# ONE

# ARRAY OBJECTS

NumPy provides an N-dimensional array type, the *ndarray*, which describes a collection of "items" of the same type. The items can be *indexed* using for example N integers.

All ndarrays are homogenous: every item takes up the same size block of memory, and all blocks are interpreted in exactly the same way. How each item in the array is to be interpreted is specified by a separate *data-type object*, one of which is associated with every array. In addition to basic types (integers, floats, *etc.*), the data type objects can also represent data structures.

An item extracted from an array, *e.g.*, by indexing, is represented by a Python object whose type is one of the *array scalar types* built in NumPy. The array scalars allow easy manipulation of also more complicated arrangements of data.

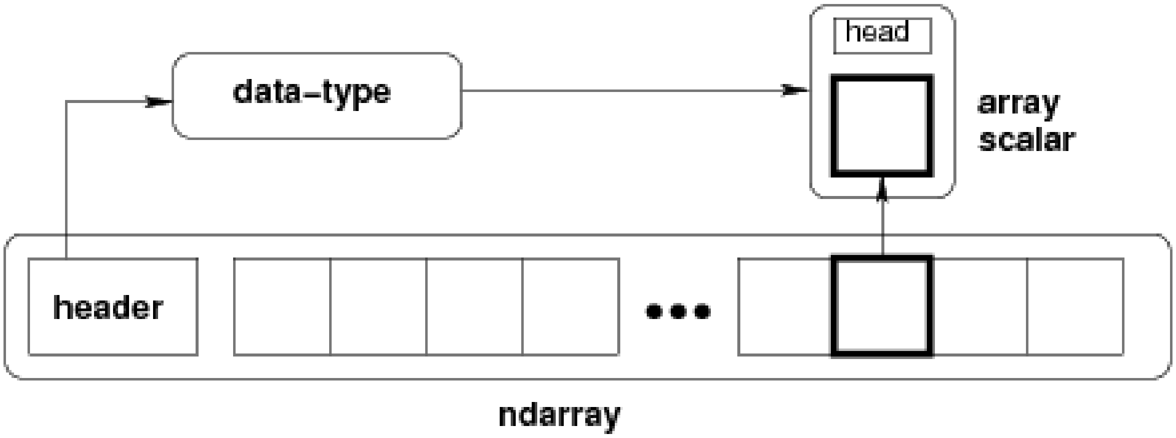


Fig. 1: **Figure** Conceptual diagram showing the relationship between the three fundamental objects used to describe the data in an array: 1) the ndarray itself, 2) the data-type object that describes the layout of a single fixed-size element of the array, 3) the array-scalar Python object that is returned when a single element of the array is accessed.

## 1.1 The N-dimensional array (`ndarray`)

An `ndarray` is a (usually fixed-size) multidimensional container of items of the same type and size. The number of dimensions and items in an array is defined by its `shape`, which is a `tuple` of *N* non-negative integers that specify the sizes of each dimension. The type of items in the array is specified by a separate *data-type object (dtype)*, one of which is associated with each ndarray.

As with other container objects in Python, the contents of an `ndarray` can be accessed and modified by *indexing or slicing* the array (using, for example, *N* integers), and via the methods and attributes of the `ndarray`.

Different `ndarrays` can share the same data, so that changes made in one `ndarray` may be visible in another. That is, an ndarray can be a *"view"* to another ndarray, and the data it is referring to is taken care of by the *"base"* ndarray. ndarrays can also be views to memory owned by Python `strings` or objects implementing the `buffer` or *array* interfaces.

---

**Example**

A 2-dimensional array of size 2 x 3, composed of 4-byte integer elements:

```
>>> x = np.array([[1, 2, 3], [4, 5, 6]], np.int32)
>>> type(x)
<class 'numpy.ndarray'>
>>> x.shape
(2, 3)
>>> x.dtype
dtype('int32')
```

The array can be indexed using Python container-like syntax:

```
>>> # The element of x in the *second* row, *third* column, namely, 6.
>>> x[1, 2]
6
```

For example *slicing* can produce views of the array:

```
>>> y = x[:,1]
>>> y
array([2, 5])
>>> y[0] = 9 # this also changes the corresponding element in x
>>> y
array([9, 5])
>>> x
array([[1, 9, 3],
       [4, 5, 6]])
```

---

## 1.1.1 Constructing arrays

New arrays can be constructed using the routines detailed in *Array creation routines*, and also by using the low-level `ndarray` constructor:

| | |
|---|---|
| `ndarray`(shape[, dtype, buffer, offset, …]) | An array object represents a multidimensional, homogeneous array of fixed-size items. |

**class** `numpy.`**`ndarray`**(*shape*, *dtype=float*, *buffer=None*, *offset=0*, *strides=None*, *order=None*)

An array object represents a multidimensional, homogeneous array of fixed-size items. An associated data-type object describes the format of each element in the array (its byte-order, how many bytes it occupies in memory, whether it is an integer, a floating point number, or something else, etc.)

Arrays should be constructed using `array`, `zeros` or `empty` (refer to the See Also section below). The parameters given here refer to a low-level method (*ndarray(…)*) for instantiating an array.

For more information, refer to the `numpy` module and examine the methods and attributes of an array.

**Parameters**

**(for the __new__ method; see Notes below)**

**shape**

[tuple of ints] Shape of created array.

**dtype**

[data-type, optional] Any object that can be interpreted as a numpy data type.

**buffer**

[object exposing buffer interface, optional] Used to fill the array with data.

**offset**

[int, optional] Offset of array data in buffer.

**strides**

[tuple of ints, optional] Strides of data in memory.

**order**

[{'C', 'F'}, optional] Row-major (C-style) or column-major (Fortran-style) order.

**See also:**

**`array`**

Construct an array.

**`zeros`**

Create an array, each element of which is zero.

**`empty`**

Create an array, but leave its allocated memory unchanged (i.e., it contains "garbage").

**`dtype`**

Create a data-type.

### Notes

There are two modes of creating an array using `__new__`:

1. If *buffer* is None, then only `shape`, `dtype`, and *order* are used.
2. If *buffer* is an object exposing the buffer interface, then all keywords are interpreted.

No `__init__` method is needed because the array is fully initialized after the `__new__` method.

### Examples

These examples illustrate the low-level `ndarray` constructor. Refer to the *See Also* section above for easier ways of constructing an ndarray.

First mode, *buffer* is None:

```
>>> np.ndarray(shape=(2,2), dtype=float, order='F')
array([[0.0e+000, 0.0e+000], # random
       [     nan, 2.5e-323]])
```

Second mode:

```
>>> np.ndarray((2,), buffer=np.array([1,2,3]),
...            offset=np.int_().itemsize,
...            dtype=int) # offset = 1*itemsize, i.e. skip first element
array([2, 3])
```

**Attributes**

***T***

[ndarray] The transposed array.

***data***

[buffer] Python buffer object pointing to the start of the array's data.

***dtype***

[dtype object] Data-type of the array's elements.

***flags***

[dict] Information about the memory layout of the array.

***flat***

[numpy.flatiter object] A 1-D iterator over the array.

***imag***

[ndarray] The imaginary part of the array.

***real***

[ndarray] The real part of the array.

***size***

[int] Number of elements in the array.

***itemsize***

[int] Length of one array element in bytes.

***nbytes***

[int] Total bytes consumed by the elements of the array.

***ndim***

[int] Number of array dimensions.

***shape***

[tuple of ints] Tuple of array dimensions.

***strides***

[tuple of ints] Tuple of bytes to step in each dimension when traversing an array.

***ctypes***

[ctypes object] An object to simplify the interaction of the array with the ctypes module.

***base***

[ndarray] Base object if memory is from some other object.

### Methods

| | |
|---|---|
| `all`([axis, out, keepdims]) | Returns True if all elements evaluate to True. |
| `any`([axis, out, keepdims]) | Returns True if any of the elements of *a* evaluate to True. |
| `argmax`([axis, out]) | Return indices of the maximum values along the given axis. |
| `argmin`([axis, out]) | Return indices of the minimum values along the given axis of *a*. |
| `argpartition`(kth[, axis, kind, order]) | Returns the indices that would partition this array. |
| `argsort`([axis, kind, order]) | Returns the indices that would sort this array. |
| `astype`(dtype[, order, casting, subok, copy]) | Copy of the array, cast to a specified type. |
| `byteswap`([inplace]) | Swap the bytes of the array elements |
| `choose`(choices[, out, mode]) | Use an index array to construct a new array from a set of choices. |
| `clip`([min, max, out]) | Return an array whose values are limited to `[min, max]`. |
| `compress`(condition[, axis, out]) | Return selected slices of this array along given axis. |
| `conj`() | Complex-conjugate all elements. |
| `conjugate`() | Return the complex conjugate, element-wise. |
| `copy`([order]) | Return a copy of the array. |
| `cumprod`([axis, dtype, out]) | Return the cumulative product of the elements along the given axis. |
| `cumsum`([axis, dtype, out]) | Return the cumulative sum of the elements along the given axis. |
| `diagonal`([offset, axis1, axis2]) | Return specified diagonals. |
| `dot`(b[, out]) | Dot product of two arrays. |
| `dump`(file) | Dump a pickle of the array to the specified file. |
| `dumps`() | Returns the pickle of the array as a string. |
| `fill`(value) | Fill the array with a scalar value. |
| `flatten`([order]) | Return a copy of the array collapsed into one dimension. |
| `getfield`(dtype[, offset]) | Returns a field of the given array as a certain type. |
| `item`(*args) | Copy an element of an array to a standard Python scalar and return it. |
| `itemset`(*args) | Insert scalar into an array (scalar is cast to array's dtype, if possible) |
| `max`([axis, out, keepdims, initial, where]) | Return the maximum along a given axis. |
| `mean`([axis, dtype, out, keepdims]) | Returns the average of the array elements along given axis. |
| `min`([axis, out, keepdims, initial, where]) | Return the minimum along a given axis. |
| `newbyteorder`([new_order]) | Return the array with the same data viewed with a different byte order. |
| `nonzero`() | Return the indices of the elements that are non-zero. |
| `partition`(kth[, axis, kind, order]) | Rearranges the elements in the array in such a way that the value of the element in kth position is in the position it would be in a sorted array. |
| `prod`([axis, dtype, out, keepdims, initial, …]) | Return the product of the array elements over the given axis |
| `ptp`([axis, out, keepdims]) | Peak to peak (maximum - minimum) value along a given axis. |

Continued on next page

Table 2 – continued from previous page

| | |
|---|---|
| `put`(indices, values[, mode]) | Set `a.flat[n] = values[n]` for all *n* in indices. |
| `ravel`([order]) | Return a flattened array. |
| `repeat`(repeats[, axis]) | Repeat elements of an array. |
| `reshape`(shape[, order]) | Returns an array containing the same data with a new shape. |
| `resize`(new_shape[, refcheck]) | Change shape and size of array in-place. |
| `round`([decimals, out]) | Return *a* with each element rounded to the given number of decimals. |
| `searchsorted`(v[, side, sorter]) | Find indices where elements of v should be inserted in a to maintain order. |
| `setfield`(val, dtype[, offset]) | Put a value into a specified place in a field defined by a data-type. |
| `setflags`([write, align, uic]) | Set array flags WRITEABLE, ALIGNED, (WRITEBACKIFCOPY and UPDATEIFCOPY), respectively. |
| `sort`([axis, kind, order]) | Sort an array in-place. |
| `squeeze`([axis]) | Remove single-dimensional entries from the shape of *a*. |
| `std`([axis, dtype, out, ddof, keepdims]) | Returns the standard deviation of the array elements along given axis. |
| `sum`([axis, dtype, out, keepdims, initial, where]) | Return the sum of the array elements over the given axis. |
| `swapaxes`(axis1, axis2) | Return a view of the array with *axis1* and *axis2* interchanged. |
| `take`(indices[, axis, out, mode]) | Return an array formed from the elements of *a* at the given indices. |
| `tobytes`([order]) | Construct Python bytes containing the raw data bytes in the array. |
| `tofile`(fid[, sep, format]) | Write array to a file as text or binary (default). |
| `tolist`() | Return the array as an `a.ndim`-levels deep nested list of Python scalars. |
| `tostring`([order]) | A compatibility alias for `tobytes`, with exactly the same behavior. |
| `trace`([offset, axis1, axis2, dtype, out]) | Return the sum along diagonals of the array. |
| `transpose`(*axes) | Returns a view of the array with axes transposed. |
| `var`([axis, dtype, out, ddof, keepdims]) | Returns the variance of the array elements, along given axis. |
| `view`([dtype][, type]) | New view of array with the same data. |

method

`ndarray.`**`all`**(*axis=None*, *out=None*, *keepdims=False*)

Returns True if all elements evaluate to True.

Refer to `numpy.all` for full documentation.

**See also:**

**`numpy.all`**

equivalent function

method

`ndarray.`**`any`**`(`*axis=None*, *out=None*, *keepdims=False*`)`

Returns True if any of the elements of *a* evaluate to True.

Refer to `numpy.any` for full documentation.

**See also:**

**`numpy.any`**

equivalent function

method

`ndarray.`**`argmax`**`(`*axis=None*, *out=None*`)`

Return indices of the maximum values along the given axis.

Refer to `numpy.argmax` for full documentation.

**See also:**

**`numpy.argmax`**

equivalent function

method

`ndarray.`**`argmin`**`(`*axis=None*, *out=None*`)`

Return indices of the minimum values along the given axis of *a*.

Refer to `numpy.argmin` for detailed documentation.

**See also:**

**`numpy.argmin`**

equivalent function

method

`ndarray.`**`argpartition`**`(`*kth*, *axis=-1*, *kind='introselect'*, *order=None*`)`

Returns the indices that would partition this array.

Refer to `numpy.argpartition` for full documentation.

New in version 1.8.0.

**See also:**

**`numpy.argpartition`**

equivalent function

method

`ndarray.`**`argsort`**`(`*axis=-1*, *kind=None*, *order=None*`)`

Returns the indices that would sort this array.

Refer to `numpy.argsort` for full documentation.

**See also:**

**`numpy.argsort`**

equivalent function

method

`ndarray.`**`astype`**(*dtype*, *order='K'*, *casting='unsafe'*, *subok=True*, *copy=True*)

Copy of the array, cast to a specified type.

**Parameters**

**dtype**

[str or dtype] Typecode or data-type to which the array is cast.

**order**

[{'C', 'F', 'A', 'K'}, optional] Controls the memory layout order of the result. 'C' means C order, 'F' means Fortran order, 'A' means 'F' order if all the arrays are Fortran contiguous, 'C' order otherwise, and 'K' means as close to the order the array elements appear in memory as possible. Default is 'K'.

**casting**

[{'no', 'equiv', 'safe', 'same_kind', 'unsafe'}, optional] Controls what kind of data casting may occur. Defaults to 'unsafe' for backwards compatibility.

- 'no' means the data types should not be cast at all.
- 'equiv' means only byte-order changes are allowed.
- 'safe' means only casts which can preserve values are allowed.
- 'same_kind' means only safe casts or casts within a kind, like float64 to float32, are allowed.
- 'unsafe' means any data conversions may be done.

**subok**

[bool, optional] If True, then sub-classes will be passed-through (default), otherwise the returned array will be forced to be a base-class array.

**copy**

[bool, optional] By default, astype always returns a newly allocated array. If this is set to false, and the `dtype`, *order*, and *subok* requirements are satisfied, the input array is returned instead of a copy.

**Returns**

**arr_t**

[ndarray] Unless `copy` is False and the other conditions for returning the input array are satisfied (see description for `copy` input parameter), *arr_t* is a new array of the same shape as the input array, with dtype, order given by `dtype`, *order*.

**Raises**

**ComplexWarning**

When casting from complex to float or int. To avoid this, one should use `a.real.astype(t)`.

## Notes

Changed in version 1.17.0: Casting between a simple data type and a structured one is possible only for "unsafe" casting. Casting to multiple fields is allowed, but casting from multiple fields is not.

Changed in version 1.9.0: Casting from numeric to string types in 'safe' casting mode requires that the string dtype length is long enough to store the max integer/float value converted.

## Examples

```
>>> x = np.array([1, 2, 2.5])
>>> x
array([1. ,  2. ,  2.5])
```

```
>>> x.astype(int)
array([1, 2, 2])
```

method

`ndarray.`**`byteswap`**`(`*inplace=False*`)`

Swap the bytes of the array elements

Toggle between low-endian and big-endian data representation by returning a byteswapped array, optionally swapped in-place. Arrays of byte-strings are not swapped. The real and imaginary parts of a complex number are swapped individually.

**Parameters**

**inplace**

[bool, optional] If `True`, swap bytes in-place, default is `False`.

**Returns**

**out**

[ndarray] The byteswapped array. If *inplace* is `True`, this is a view to self.

## Examples

```
>>> A = np.array([1, 256, 8755], dtype=np.int16)
>>> list(map(hex, A))
['0x1', '0x100', '0x2233']
>>> A.byteswap(inplace=True)
array([  256,     1, 13090], dtype=int16)
>>> list(map(hex, A))
['0x100', '0x1', '0x3322']
```

Arrays of byte-strings are not swapped

```
>>> A = np.array([b'ceg', b'fac'])
>>> A.byteswap()
array([b'ceg', b'fac'], dtype='|S3')
```

**`A.newbyteorder().byteswap()` produces an array with the same values**

but different representation in memory

```
>>> A = np.array([1, 2, 3])
>>> A.view(np.uint8)
array([1, 0, 0, 0, 0, 0, 0, 0, 2, 0, 0, 0, 0, 0, 0, 0, 3, 0, 0, 0, 0, 0,
       0, 0], dtype=uint8)
>>> A.newbyteorder().byteswap(inplace=True)
array([1, 2, 3])
>>> A.view(np.uint8)
array([0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 2, 0, 0, 0, 0, 0, 0,
       0, 3], dtype=uint8)
```

method

`ndarray.`**`choose`**(*choices*, *out=None*, *mode='raise'*)

Use an index array to construct a new array from a set of choices.

Refer to *`numpy.choose`* for full documentation.

**See also:**

***`numpy.choose`***

equivalent function

method

`ndarray.`**`clip`**(*min=None*, *max=None*, *out=None*, ***kwargs*)

Return an array whose values are limited to `[min, max]`. One of max or min must be given.

Refer to *`numpy.clip`* for full documentation.

**See also:**

***`numpy.clip`***

equivalent function

method

`ndarray.`**`compress`**(*condition*, *axis=None*, *out=None*)

Return selected slices of this array along given axis.

Refer to *`numpy.compress`* for full documentation.

**See also:**

***`numpy.compress`***

equivalent function

method

`ndarray.`**`conj`**()

Complex-conjugate all elements.

Refer to *`numpy.conjugate`* for full documentation.

**See also:**

**`numpy.conjugate`**

equivalent function

method

`ndarray.`**`conjugate`**`()`

Return the complex conjugate, element-wise.

Refer to `numpy.conjugate` for full documentation.

**See also:**

**`numpy.conjugate`**

equivalent function

method

`ndarray.`**`copy`**`(`*order='C'*`)`

Return a copy of the array.

**Parameters**

**order**

[{'C', 'F', 'A', 'K'}, optional] Controls the memory layout of the copy. 'C' means C-order, 'F' means F-order, 'A' means 'F' if *a* is Fortran contiguous, 'C' otherwise. 'K' means match the layout of *a* as closely as possible. (Note that this function and `numpy.copy` are very similar, but have different default values for their order= arguments.)

**See also:**

`numpy.copy`, `numpy.copyto`

### Examples

```
>>> x = np.array([[1,2,3],[4,5,6]], order='F')
```

```
>>> y = x.copy()
```

```
>>> x.fill(0)
```

```
>>> x
array([[0, 0, 0],
       [0, 0, 0]])
```

```
>>> y
array([[1, 2, 3],
       [4, 5, 6]])
```

```
>>> y.flags['C_CONTIGUOUS']
True
```

method

`ndarray.`**`cumprod`**`(`*axis=None*, *dtype=None*, *out=None*`)`

Return the cumulative product of the elements along the given axis.

Refer to `numpy.cumprod` for full documentation.

**See also:**

***`numpy.cumprod`***

equivalent function

method

`ndarray.`**`cumsum`**`(`*axis=None*, *dtype=None*, *out=None*`)`

Return the cumulative sum of the elements along the given axis.

Refer to `numpy.cumsum` for full documentation.

**See also:**

***`numpy.cumsum`***

equivalent function

method

`ndarray.`**`diagonal`**`(`*offset=0*, *axis1=0*, *axis2=1*`)`

Return specified diagonals. In NumPy 1.9 the returned array is a read-only view instead of a copy as in previous NumPy versions. In a future version the read-only restriction will be removed.

Refer to `numpy.diagonal` for full documentation.

**See also:**

***`numpy.diagonal`***

equivalent function

method

`ndarray.`**`dot`**`(`*b*, *out=None*`)`

Dot product of two arrays.

Refer to `numpy.dot` for full documentation.

**See also:**

***`numpy.dot`***

equivalent function

### Examples

```
>>> a = np.eye(2)
>>> b = np.ones((2, 2)) * 2
>>> a.dot(b)
array([[2.,  2.],
       [2.,  2.]])
```

This array method can be conveniently chained:

```
>>> a.dot(b).dot(b)
array([[8.,  8.],
       [8.,  8.]])
```

method

`ndarray.`**`dump`**`(`*file*`)`

Dump a pickle of the array to the specified file. The array can be read back with pickle.load or numpy.load.

**Parameters**

**file**

[str or Path] A string naming the dump file.

Changed in version 1.17.0: `pathlib.Path` objects are now accepted.

method

`ndarray.`**`dumps`**`()`

Returns the pickle of the array as a string. pickle.loads or numpy.loads will convert the string back to an array.

**Parameters**

**None**

method

`ndarray.`**`fill`**`(`*value*`)`

Fill the array with a scalar value.

**Parameters**

**value**

[scalar] All elements of *a* will be assigned this value.

### Examples

```
>>> a = np.array([1, 2])
>>> a.fill(0)
>>> a
array([0, 0])
>>> a = np.empty(2)
>>> a.fill(1)
>>> a
array([1.,  1.])
```

method

`ndarray.`**`flatten`**`(`*order='C'*`)`

Return a copy of the array collapsed into one dimension.

**Parameters**

**order**

[{'C', 'F', 'A', 'K'}, optional] 'C' means to flatten in row-major (C-style) order. 'F' means to flatten in column-major (Fortran- style) order. 'A' means to flatten in column-major order if *a* is Fortran *contiguous* in memory, row-major order otherwise. 'K' means to flatten *a* in the order the elements occur in memory. The default is 'C'.

**Returns**

**y**

[ndarray] A copy of the input array, flattened to one dimension.

**See also:**

***`ravel`***

Return a flattened array.

***`flat`***

A 1-D flat iterator over the array.

### Examples

```
>>> a = np.array([[1,2], [3,4]])
>>> a.flatten()
array([1, 2, 3, 4])
>>> a.flatten('F')
array([1, 3, 2, 4])
```

method

`ndarray.`**`getfield`**`(`*dtype*, *offset=0*`)`

Returns a field of the given array as a certain type.

A field is a view of the array data with a given data-type. The values in the view are determined by the given type and the offset into the current array in bytes. The offset needs to be such that the view dtype fits in the array dtype; for example an array of dtype complex128 has 16-byte elements. If taking a view with a 32-bit integer (4 bytes), the offset needs to be between 0 and 12 bytes.

**Parameters**

**dtype**

[str or dtype] The data type of the view. The dtype size of the view can not be larger than that of the array itself.

**offset**

[int] Number of bytes to skip before beginning the element view.

## Examples

```
>>> x = np.diag([1.+1.j]*2)
>>> x[1, 1] = 2 + 4.j
>>> x
array([[1.+1.j,  0.+0.j],
       [0.+0.j,  2.+4.j]])
>>> x.getfield(np.float64)
array([[1.,  0.],
       [0.,  2.]])
```

By choosing an offset of 8 bytes we can select the complex part of the array for our view:

```
>>> x.getfield(np.float64, offset=8)
array([[1.,  0.],
       [0.,  4.]])
```

method

`ndarray.`**`item`**`(`**args*`)`

Copy an element of an array to a standard Python scalar and return it.

**Parameters**

***args**

[Arguments (variable number and type)]

- none: in this case, the method only works for arrays with one element (*a.size == 1*), which element is copied into a standard Python scalar object and returned.
- int_type: this argument is interpreted as a flat index into the array, specifying which element to copy and return.
- tuple of int_types: functions as does a single int_type argument, except that the argument is interpreted as an nd-index into the array.

**Returns**

**z**

[Standard Python scalar object] A copy of the specified element of the array as a suitable Python scalar

## Notes

When the data type of *a* is longdouble or clongdouble, item() returns a scalar array object because there is no available Python scalar that would not lose information. Void arrays return a buffer object for item(), unless fields are defined, in which case a tuple is returned.

`item` is very similar to a[args], except, instead of an array scalar, a standard Python scalar is returned. This can be useful for speeding up access to elements of the array and doing arithmetic on elements of the array using Python's optimized math.

## Examples

```
>>> np.random.seed(123)
>>> x = np.random.randint(9, size=(3, 3))
>>> x
array([[2, 2, 6],
       [1, 3, 6],
       [1, 0, 1]])
>>> x.item(3)
1
>>> x.item(7)
0
>>> x.item((0, 1))
2
>>> x.item((2, 2))
1
```

method

`ndarray.`**`itemset`**`(`**args*`)`

Insert scalar into an array (scalar is cast to array's dtype, if possible)

There must be at least 1 argument, and define the last argument as *item*. Then, `a.itemset(*args)` is equivalent to but faster than `a[args] = item`. The item should be a scalar value and *args* must select a single item in the array *a*.

**Parameters**

***args**

[Arguments] If one argument: a scalar, only used in case *a* is of size 1. If two arguments: the last argument is the value to be set and must be a scalar, the first argument specifies a single array element location. It is either an int or a tuple.

## Notes

Compared to indexing syntax, *`itemset`* provides some speed increase for placing a scalar into a particular location in an *`ndarray`*, if you must do this. However, generally this is discouraged: among other problems, it complicates the appearance of the code. Also, when using *`itemset`* (and *`item`*) inside a loop, be sure to assign the methods to a local variable to avoid the attribute look-up at each loop iteration.

## Examples

```
>>> np.random.seed(123)
>>> x = np.random.randint(9, size=(3, 3))
>>> x
array([[2, 2, 6],
       [1, 3, 6],
       [1, 0, 1]])
>>> x.itemset(4, 0)
>>> x.itemset((2, 2), 9)
>>> x
array([[2, 2, 6],
       [1, 0, 6],
       [1, 0, 9]])
```

method

`ndarray.`**`max`**(*axis=None*, *out=None*, *keepdims=False*, *initial=<no value>*, *where=True*)

Return the maximum along a given axis.

Refer to *`numpy.amax`* for full documentation.

**See also:**

***`numpy.amax`***

equivalent function

method

`ndarray.`**`mean`**(*axis=None*, *dtype=None*, *out=None*, *keepdims=False*)

Returns the average of the array elements along given axis.

Refer to *`numpy.mean`* for full documentation.

**See also:**

***`numpy.mean`***

equivalent function

method

`ndarray.`**`min`**(*axis=None*, *out=None*, *keepdims=False*, *initial=<no value>*, *where=True*)

Return the minimum along a given axis.

Refer to *`numpy.amin`* for full documentation.

**See also:**

***`numpy.amin`***

equivalent function

method

`ndarray.`**`newbyteorder`**(*new_order='S'*)

Return the array with the same data viewed with a different byte order.

Equivalent to:

```
arr.view(arr.dtype.newbytorder(new_order))
```

Changes are also made in all fields and sub-arrays of the array data type.

**Parameters**

**new_order**

[string, optional] Byte order to force; a value from the byte order specifications below. *new_order* codes can be any of:

- 'S' - swap dtype from current to opposite endian
- {'<', 'L'} - little endian
- {'>', 'B'} - big endian
- {'=', 'N'} - native order

- {'|', 'I'} - ignore (no change to byte order)

The default value ('S') results in swapping the current byte order. The code does a case-insensitive check on the first letter of *new_order* for the alternatives above. For example, any of 'B' or 'b' or 'biggish' are valid to specify big-endian.

**Returns**

**new_arr**

[array] New array object with the dtype reflecting given change to the byte order.

method

`ndarray.`**`nonzero`**`()`

Return the indices of the elements that are non-zero.

Refer to `numpy.nonzero` for full documentation.

**See also:**

**`numpy.nonzero`**

equivalent function

method

`ndarray.`**`partition`**`(`*kth*, *axis=-1*, *kind='introselect'*, *order=None*`)`

Rearranges the elements in the array in such a way that the value of the element in kth position is in the position it would be in a sorted array. All elements smaller than the kth element are moved before this element and all equal or greater are moved behind it. The ordering of the elements in the two partitions is undefined.

New in version 1.8.0.

**Parameters**

**kth**

[int or sequence of ints] Element index to partition by. The kth element value will be in its final sorted position and all smaller elements will be moved before it and all equal or greater elements behind it. The order of all elements in the partitions is undefined. If provided with a sequence of kth it will partition all elements indexed by kth of them into their sorted position at once.

**axis**

[int, optional] Axis along which to sort. Default is -1, which means sort along the last axis.

**kind**

[{'introselect'}, optional] Selection algorithm. Default is 'introselect'.

**order**

[str or list of str, optional] When *a* is an array with fields defined, this argument specifies which fields to compare first, second, etc. A single field can be specified as a string, and not all fields need to be specified, but unspecified fields will still be used, in the order in which they come up in the dtype, to break ties.

**See also:**

***numpy.partition***

Return a parititioned copy of an array.

***argpartition***

Indirect partition.

***sort***

Full sort.

### Notes

See `np.partition` for notes on the different algorithms.

### Examples

```
>>> a = np.array([3, 4, 2, 1])
>>> a.partition(3)
>>> a
array([2, 1, 3, 4])
```

```
>>> a.partition((1, 3))
>>> a
array([1, 2, 3, 4])
```

method

`ndarray.`**`prod`**(*axis=None*, *dtype=None*, *out=None*, *keepdims=False*, *initial=1*, *where=True*)

Return the product of the array elements over the given axis

Refer to *`numpy.prod`* for full documentation.

**See also:**

***numpy.prod***

equivalent function

method

`ndarray.`**`ptp`**(*axis=None*, *out=None*, *keepdims=False*)

Peak to peak (maximum - minimum) value along a given axis.

Refer to *`numpy.ptp`* for full documentation.

**See also:**

***numpy.ptp***

equivalent function

method

`ndarray.`**`put`**(*indices*, *values*, *mode='raise'*)

Set `a.flat[n] = values[n]` for all *n* in indices.

Refer to *`numpy.put`* for full documentation.

**See also:**

**`numpy.put`**

equivalent function

method

`ndarray.`**`ravel`**`(`[*order*]`)`

Return a flattened array.

Refer to `numpy.ravel` for full documentation.

**See also:**

**`numpy.ravel`**

equivalent function

**`ndarray.flat`**

a flat iterator on the array.

method

`ndarray.`**`repeat`**`(`*repeats*, *axis=None*`)`

Repeat elements of an array.

Refer to `numpy.repeat` for full documentation.

**See also:**

**`numpy.repeat`**

equivalent function

method

`ndarray.`**`reshape`**`(`*shape*, *order='C'*`)`

Returns an array containing the same data with a new shape.

Refer to `numpy.reshape` for full documentation.

**See also:**

**`numpy.reshape`**

equivalent function

### Notes

Unlike the free function `numpy.reshape`, this method on `ndarray` allows the elements of the shape parameter to be passed in as separate arguments. For example, `a.reshape(10, 11)` is equivalent to `a.reshape((10, 11))`.

method

`ndarray.`**`resize`**`(`*new_shape*, *refcheck=True*`)`

Change shape and size of array in-place.

**Parameters**

**new_shape**

[tuple of ints, or *n* ints] Shape of resized array.

**refcheck**

[bool, optional] If False, reference count will not be checked. Default is True.

**Returns**

**None**

**Raises**

**ValueError**

If *a* does not own its own data or references or views to it exist, and the data memory must be changed. PyPy only: will always raise if the data memory must be changed, since there is no reliable way to determine if references or views to it exist.

**SystemError**

If the *order* keyword argument is specified. This behaviour is a bug in NumPy.

**See also:**

**`resize`**

Return a new array with the specified shape.

## Notes

This reallocates space for the data area if necessary.

Only contiguous arrays (data elements consecutive in memory) can be resized.

The purpose of the reference count check is to make sure you do not use this array as a buffer for another Python object and then reallocate the memory. However, reference counts can increase in other ways so if you are sure that you have not shared the memory for this array with another Python object, then you may safely set *refcheck* to False.

## Examples

Shrinking an array: array is flattened (in the order that the data are stored in memory), resized, and reshaped:

```
>>> a = np.array([[0, 1], [2, 3]], order='C')
>>> a.resize((2, 1))
>>> a
array([[0],
       [1]])
```

```
>>> a = np.array([[0, 1], [2, 3]], order='F')
>>> a.resize((2, 1))
>>> a
array([[0],
       [2]])
```

Enlarging an array: as above, but missing entries are filled with zeros:

```
>>> b = np.array([[0, 1], [2, 3]])
>>> b.resize(2, 3) # new_shape parameter doesn't have to be a tuple
>>> b
array([[0, 1, 2],
       [3, 0, 0]])
```

Referencing an array prevents resizing…

```
>>> c = a
>>> a.resize((1, 1))
Traceback (most recent call last):
...
ValueError: cannot resize an array that references or is referenced ...
```

Unless *refcheck* is False:

```
>>> a.resize((1, 1), refcheck=False)
>>> a
array([[0]])
>>> c
array([[0]])
```

method

`ndarray.`**`round`**(*decimals=0*, *out=None*)

Return *a* with each element rounded to the given number of decimals.

Refer to `numpy.around` for full documentation.

**See also:**

**`numpy.around`**

equivalent function

method

`ndarray.`**`searchsorted`**(*v*, *side='left'*, *sorter=None*)

Find indices where elements of v should be inserted in a to maintain order.

For full documentation, see `numpy.searchsorted`

**See also:**

**`numpy.searchsorted`**

equivalent function

method

`ndarray.`**`setfield`**(*val*, *dtype*, *offset=0*)

Put a value into a specified place in a field defined by a data-type.

Place *val* into *a*'s field defined by `dtype` and beginning *offset* bytes into the field.

**Parameters**

**val**

[object] Value to be placed in field.

**dtype**

[dtype object] Data-type of the field in which to place *val*.

**offset**

[int, optional] The number of bytes into the field at which to place *val*.

**Returns**

**None**

**See also:**

`getfield`

### Examples

```
>>> x = np.eye(3)
>>> x.getfield(np.float64)
array([[1.,  0.,  0.],
       [0.,  1.,  0.],
       [0.,  0.,  1.]])
>>> x.setfield(3, np.int32)
>>> x.getfield(np.int32)
array([[3, 3, 3],
       [3, 3, 3],
       [3, 3, 3]], dtype=int32)
>>> x
array([[1.0e+000, 1.5e-323, 1.5e-323],
       [1.5e-323, 1.0e+000, 1.5e-323],
       [1.5e-323, 1.5e-323, 1.0e+000]])
>>> x.setfield(np.eye(3), np.int32)
>>> x
array([[1.,  0.,  0.],
       [0.,  1.,  0.],
       [0.,  0.,  1.]])
```

method

`ndarray.`**`setflags`**(*write=None*, *align=None*, *uic=None*)

Set array flags WRITEABLE, ALIGNED, (WRITEBACKIFCOPY and UPDATEIFCOPY), respectively.

These Boolean-valued flags affect how numpy interprets the memory area used by *a* (see Notes below). The ALIGNED flag can only be set to True if the data is actually aligned according to the type. The WRITEBACKIFCOPY and (deprecated) UPDATEIFCOPY flags can never be set to True. The flag WRITEABLE can only be set to True if the array owns its own memory, or the ultimate owner of the memory exposes a writeable buffer interface, or is a string. (The exception for string is made so that unpickling can be done without copying memory.)

**Parameters**

**write**

[bool, optional] Describes whether or not *a* can be written to.

**align**

[bool, optional] Describes whether or not *a* is aligned properly for its type.

**uic**

[bool, optional] Describes whether or not *a* is a copy of another "base" array.

### Notes

Array flags provide information about how the memory area used for the array is to be interpreted. There are 7 Boolean flags in use, only four of which can be changed by the user: WRITEBACKIFCOPY, UPDATEIFCOPY, WRITEABLE, and ALIGNED.

WRITEABLE (W) the data area can be written to;

ALIGNED (A) the data and strides are aligned appropriately for the hardware (as determined by the compiler);

UPDATEIFCOPY (U) (deprecated), replaced by WRITEBACKIFCOPY;

WRITEBACKIFCOPY (X) this array is a copy of some other array (referenced by .base). When the C-API function PyArray_ResolveWritebackIfCopy is called, the base array will be updated with the contents of this array.

All flags can be accessed using the single (upper case) letter as well as the full name.

### Examples

```
>>> y = np.array([[3, 1, 7],
...               [2, 0, 0],
...               [8, 5, 9]])
>>> y
array([[3, 1, 7],
       [2, 0, 0],
       [8, 5, 9]])
>>> y.flags
  C_CONTIGUOUS : True
  F_CONTIGUOUS : False
  OWNDATA : True
  WRITEABLE : True
  ALIGNED : True
  WRITEBACKIFCOPY : False
  UPDATEIFCOPY : False
>>> y.setflags(write=0, align=0)
>>> y.flags
  C_CONTIGUOUS : True
  F_CONTIGUOUS : False
  OWNDATA : True
  WRITEABLE : False
  ALIGNED : False
  WRITEBACKIFCOPY : False
  UPDATEIFCOPY : False
>>> y.setflags(uic=1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: cannot set WRITEBACKIFCOPY flag to True
```

method

`ndarray.`**`sort`**(*axis=-1*, *kind=None*, *order=None*)

Sort an array in-place. Refer to `numpy.sort` for full documentation.

**Parameters**

**axis**

[int, optional] Axis along which to sort. Default is -1, which means sort along the last axis.

**kind**

[{'quicksort', 'mergesort', 'heapsort', 'stable'}, optional] Sorting algorithm. The default is 'quicksort'. Note that both 'stable' and 'mergesort' use timsort under the covers and, in general, the actual implementation will vary with datatype. The 'mergesort' option is retained for backwards compatibility.

Changed in version 1.15.0.: The 'stable' option was added.

**order**

[str or list of str, optional] When *a* is an array with fields defined, this argument specifies which fields to compare first, second, etc. A single field can be specified as a string, and not all fields need be specified, but unspecified fields will still be used, in the order in which they come up in the dtype, to break ties.

**See also:**

**`numpy.sort`**

Return a sorted copy of an array.

**`numpy.argsort`**

Indirect sort.

**`numpy.lexsort`**

Indirect stable sort on multiple keys.

**`numpy.searchsorted`**

Find elements in sorted array.

**`numpy.partition`**

Partial sort.

## Notes

See `numpy.sort` for notes on the different sorting algorithms.

## Examples

```
>>> a = np.array([[1,4], [3,1]])
>>> a.sort(axis=1)
>>> a
array([[1, 4],
       [1, 3]])
>>> a.sort(axis=0)
>>> a
array([[1, 3],
       [1, 4]])
```

Use the *order* keyword to specify a field to use when sorting a structured array:

```
>>> a = np.array([('a', 2), ('c', 1)], dtype=[('x', 'S1'), ('y', int)])
>>> a.sort(order='y')
>>> a
array([(b'c', 1), (b'a', 2)],
      dtype=[('x', 'S1'), ('y', '<i8')])
```

method

`ndarray.`**`squeeze`**(*axis=None*)

Remove single-dimensional entries from the shape of *a*.

Refer to `numpy.squeeze` for full documentation.

**See also:**

***`numpy.squeeze`***

equivalent function

method

`ndarray.`**`std`**(*axis=None*, *dtype=None*, *out=None*, *ddof=0*, *keepdims=False*)

Returns the standard deviation of the array elements along given axis.

Refer to `numpy.std` for full documentation.

**See also:**

***`numpy.std`***

equivalent function

method

`ndarray.`**`sum`**(*axis=None*, *dtype=None*, *out=None*, *keepdims=False*, *initial=0*, *where=True*)

Return the sum of the array elements over the given axis.

Refer to `numpy.sum` for full documentation.

**See also:**

***`numpy.sum`***

equivalent function

method

`ndarray.`**`swapaxes`**(*axis1*, *axis2*)

Return a view of the array with *axis1* and *axis2* interchanged.

Refer to `numpy.swapaxes` for full documentation.

**See also:**

***`numpy.swapaxes`***

equivalent function

method

`ndarray.`**`take`**(*indices*, *axis=None*, *out=None*, *mode='raise'*)

Return an array formed from the elements of *a* at the given indices.

Refer to `numpy.take` for full documentation.

**See also:**

**`numpy.take`**

equivalent function

method

`ndarray.`**`tobytes`**(*order='C'*)

Construct Python bytes containing the raw data bytes in the array.

Constructs Python bytes showing a copy of the raw contents of data memory. The bytes object can be produced in either 'C' or 'Fortran', or 'Any' order (the default is 'C'-order). 'Any' order means C-order unless the F_CONTIGUOUS flag in the array is set, in which case it means 'Fortran' order.

New in version 1.9.0.

**Parameters**

**order**

[{'C', 'F', None}, optional] Order of the data for multidimensional arrays: C, Fortran, or the same as for the original array.

**Returns**

**s**

[bytes] Python bytes exhibiting a copy of *a*'s raw data.

### Examples

```
>>> x = np.array([[0, 1], [2, 3]], dtype='<u2')
>>> x.tobytes()
b'\x00\x00\x01\x00\x02\x00\x03\x00'
>>> x.tobytes('C') == x.tobytes()
True
>>> x.tobytes('F')
b'\x00\x00\x02\x00\x01\x00\x03\x00'
```

method

`ndarray.`**`tofile`**(*fid*, *sep=""*, *format="%s"*)

Write array to a file as text or binary (default).

Data is always written in 'C' order, independent of the order of *a*. The data produced by this method can be recovered using the function fromfile().

**Parameters**

**fid**

[file or str or Path] An open file object, or a string containing a filename.

Changed in version 1.17.0: `pathlib.Path` objects are now accepted.

**sep**

[str] Separator between array items for text output. If “” (empty), a binary file is written, equivalent to `file.write(a.tobytes())`.

**format**

[str] Format string for text file output. Each entry in the array is formatted to text by first converting it to the closest Python type, and then using “format” % item.

### Notes

This is a convenience function for quick storage of array data. Information on endianness and precision is lost, so this method is not a good choice for files intended to archive data or transport data between machines with different endianness. Some of these problems can be overcome by outputting the data as text files, at the expense of speed and file size.

When fid is a file object, array contents are directly written to the file, bypassing the file object’s `write` method. As a result, tofile cannot be used with files objects supporting compression (e.g., GzipFile) or file-like objects that do not support `fileno()` (e.g., BytesIO).

method

`ndarray.`**`tolist`**`()`

Return the array as an `a.ndim`-levels deep nested list of Python scalars.

Return a copy of the array data as a (nested) Python list. Data items are converted to the nearest compatible builtin Python type, via the *`item`* function.

If `a.ndim` is 0, then since the depth of the nested list is 0, it will not be a list at all, but a simple Python scalar.

**Parameters**

**none**

**Returns**

**y**

[object, or list of object, or list of list of object, or …] The possibly nested list of array elements.

### Notes

The array may be recreated via `a = np.array(a.tolist())`, although this may sometimes lose precision.

### Examples

For a 1D array, `a.tolist()` is almost the same as `list(a)`, except that `tolist` changes numpy scalars to Python scalars:

```
>>> a = np.uint32([1, 2])
>>> a_list = list(a)
>>> a_list
[1, 2]
>>> type(a_list[0])
<class 'numpy.uint32'>
>>> a_tolist = a.tolist()
>>> a_tolist
[1, 2]
>>> type(a_tolist[0])
<class 'int'>
```

Additionally, for a 2D array, `tolist` applies recursively:

```
>>> a = np.array([[1, 2], [3, 4]])
>>> list(a)
[array([1, 2]), array([3, 4])]
>>> a.tolist()
[[1, 2], [3, 4]]
```

The base case for this recursion is a 0D array:

```
>>> a = np.array(1)
>>> list(a)
Traceback (most recent call last):
  ...
TypeError: iteration over a 0-d array
>>> a.tolist()
1
```

method

`ndarray.`**`tostring`**`(`*order='C'*`)`

A compatibility alias for *`tobytes`*, with exactly the same behavior.

Despite its name, it returns *bytes* not `str`s.

Deprecated since version 1.19.0.

method

`ndarray.`**`trace`**`(`*offset=0*, *axis1=0*, *axis2=1*, *dtype=None*, *out=None*`)`

Return the sum along diagonals of the array.

Refer to *`numpy.trace`* for full documentation.

**See also:**

***`numpy.trace`***

equivalent function

method

`ndarray.`**`transpose`**`(`**axes*`)`

Returns a view of the array with axes transposed.

For a 1-D array this has no effect, as a transposed vector is simply the same vector. To convert a 1-D array into a 2D column vector, an additional dimension must be added. *np.atleast2d(a).T* achieves this, as does *a[:, np.newaxis]*. For a 2-D array, this is a standard matrix transpose. For an n-D array, if axes are given, their order indicates how the axes are permuted (see Examples). If axes are not provided and `a.shape = (i[0], i[1], ... i[n-2], i[n-1])`, then `a.transpose().shape = (i[n-1], i[n-2], ... i[1], i[0])`.

**Parameters**

**axes**

[None, tuple of ints, or *n* ints]

- None or no argument: reverses the order of the axes.
- tuple of ints: *i* in the *j*-th place in the tuple means *a*'s *i*-th axis becomes *a.transpose()*'s *j*-th axis.
- *n* ints: same as an n-tuple of the same ints (this form is intended simply as a "convenience" alternative to the tuple form)

**Returns**

**out**

[ndarray] View of *a*, with axes suitably permuted.

**See also:**

**`ndarray.T`**

Array property returning the array transposed.

**`ndarray.reshape`**

Give a new shape to an array without changing its data.

## Examples

```
>>> a = np.array([[1, 2], [3, 4]])
>>> a
array([[1, 2],
       [3, 4]])
>>> a.transpose()
array([[1, 3],
       [2, 4]])
>>> a.transpose((1, 0))
array([[1, 3],
       [2, 4]])
>>> a.transpose(1, 0)
array([[1, 3],
       [2, 4]])
```

method

`ndarray.`**`var`**(*axis=None*, *dtype=None*, *out=None*, *ddof=0*, *keepdims=False*)

Returns the variance of the array elements, along given axis.

Refer to `numpy.var` for full documentation.

**See also:**

*`numpy.var`*

equivalent function

method

`ndarray.`**`view`**`(`*[dtype][, type]*`)`

New view of array with the same data.

---

**Note:** Passing None for `dtype` is different from omitting the parameter, since the former invokes `dtype(None)` which is an alias for `dtype('float_')`.

---

**Parameters**

**dtype**

[data-type or ndarray sub-class, optional] Data-type descriptor of the returned view, e.g., float32 or int16. Omitting it results in the view having the same data-type as *a*. This argument can also be specified as an ndarray sub-class, which then specifies the type of the returned object (this is equivalent to setting the `type` parameter).

**type**

[Python type, optional] Type of the returned view, e.g., ndarray or matrix. Again, omission of the parameter results in type preservation.

## Notes

`a.view()` is used two different ways:

`a.view(some_dtype)` or `a.view(dtype=some_dtype)` constructs a view of the array's memory with a different data-type. This can cause a reinterpretation of the bytes of memory.

`a.view(ndarray_subclass)` or `a.view(type=ndarray_subclass)` just returns an instance of *ndarray_subclass* that looks at the same array (same shape, dtype, etc.) This does not cause a reinterpretation of the memory.

For `a.view(some_dtype)`, if `some_dtype` has a different number of bytes per entry than the previous dtype (for example, converting a regular array to a structured array), then the behavior of the view cannot be predicted just from the superficial appearance of `a` (shown by `print(a)`). It also depends on exactly how `a` is stored in memory. Therefore if `a` is C-ordered versus fortran-ordered, versus defined as a slice or transpose, etc., the view may give different results.

### Examples

```
>>> x = np.array([(1, 2)], dtype=[('a', np.int8), ('b', np.int8)])
```

Viewing array data using a different type and dtype:

```
>>> y = x.view(dtype=np.int16, type=np.matrix)
>>> y
matrix([[513]], dtype=int16)
>>> print(type(y))
<class 'numpy.matrix'>
```

Creating a view on a structured array so it can be used in calculations

```
>>> x = np.array([(1, 2),(3,4)], dtype=[('a', np.int8), ('b', np.int8)])
>>> xv = x.view(dtype=np.int8).reshape(-1,2)
>>> xv
array([[1, 2],
       [3, 4]], dtype=int8)
>>> xv.mean(0)
array([2.,  3.])
```

Making changes to the view changes the underlying array

```
>>> xv[0,1] = 20
>>> x
array([(1, 20), (3,  4)], dtype=[('a', 'i1'), ('b', 'i1')])
```

Using a view to convert an array to a recarray:

```
>>> z = x.view(np.recarray)
>>> z.a
array([1, 3], dtype=int8)
```

Views share data:

```
>>> x[0] = (9, 10)
>>> z[0]
(9, 10)
```

Views that change the dtype size (bytes per entry) should normally be avoided on arrays defined by slices, transposes, fortran-ordering, etc.:

```
>>> x = np.array([[1,2,3],[4,5,6]], dtype=np.int16)
>>> y = x[:, 0:2]
>>> y
array([[1, 2],
       [4, 5]], dtype=int16)
>>> y.view(dtype=[('width', np.int16), ('length', np.int16)])
Traceback (most recent call last):
    ...
ValueError: To change to a dtype of a different size, the array must be C-
↪contiguous
>>> z = y.copy()
>>> z.view(dtype=[('width', np.int16), ('length', np.int16)])
array([[(1, 2)],
       [(4, 5)]], dtype=[('width', '<i2'), ('length', '<i2')])
```

### 1.1.2 Indexing arrays

Arrays can be indexed using an extended Python slicing syntax, `array[selection]`. Similar syntax is also used for accessing fields in a structured data type.

**See also:**

*Array Indexing*.

### 1.1.3 Internal memory layout of an ndarray

An instance of class `ndarray` consists of a contiguous one-dimensional segment of computer memory (owned by the array, or by some other object), combined with an indexing scheme that maps *N* integers into the location of an item in the block. The ranges in which the indices can vary is specified by the `shape` of the array. How many bytes each item takes and how the bytes are interpreted is defined by the *data-type object* associated with the array.

A segment of memory is inherently 1-dimensional, and there are many different schemes for arranging the items of an *N*-dimensional array in a 1-dimensional block. NumPy is flexible, and `ndarray` objects can accommodate any *strided indexing scheme*. In a strided scheme, the N-dimensional index $(n_0, n_1, ..., n_{N-1})$ corresponds to the offset (in bytes):

$$n_{\mathrm{offset}} = \sum_{k=0}^{N-1} s_k n_k$$

from the beginning of the memory block associated with the array. Here, $s_k$ are integers which specify the `strides` of the array. The column-major order (used, for example, in the Fortran language and in *Matlab*) and row-major order (used in C) schemes are just specific kinds of strided scheme, and correspond to memory that can be *addressed* by the strides:

$$s_k^{\mathrm{column}} = \mathrm{itemsize} \prod_{j=0}^{k-1} d_j , \quad s_k^{\mathrm{row}} = \mathrm{itemsize} \prod_{j=k+1}^{N-1} d_j .$$

where $d_j$ = *self.shape[j]*.

Both the C and Fortran orders are contiguous, *i.e.,* single-segment, memory layouts, in which every part of the memory block can be accessed by some combination of the indices.

**Note:** *Contiguous arrays* and *single-segment arrays* are synonymous and are used interchangeably throughout the documentation.

While a C-style and Fortran-style contiguous array, which has the corresponding flags set, can be addressed with the above strides, the actual strides may be different. This can happen in two cases:

1. If `self.shape[k] == 1` then for any legal index `index[k] == 0`. This means that in the formula for the offset $n_k = 0$ and thus $s_k n_k = 0$ and the value of $s_k$ = *self.strides[k]* is arbitrary.
2. If an array has no elements (`self.size == 0`) there is no legal index and the strides are never used. Any array with no elements may be considered C-style and Fortran-style contiguous.

Point 1. means that `self` and `self.squeeze()` always have the same contiguity and `aligned` flags value. This also means that even a high dimensional array could be C-style and Fortran-style contiguous at the same time.

An array is considered aligned if the memory offsets for all elements and the base offset itself is a multiple of *self.itemsize*. Understanding *memory-alignment* leads to better performance on most hardware.

**Note:** Points (1) and (2) can currently be disabled by the compile time environmental variable `NPY_RELAXED_STRIDES_CHECKING=0`, which was the default before NumPy 1.10. No users should have to do this. `NPY_RELAXED_STRIDES_DEBUG=1` can be used to help find errors when incorrectly relying on the strides in C-extension code (see below warning).

You can check whether this option was enabled when your NumPy was built by looking at the value of `np.ones((10, 1), order='C').flags.f_contiguous`. If this is `True`, then your NumPy has relaxed strides checking enabled.

**Warning:** It does *not* generally hold that `self.strides[-1] == self.itemsize` for C-style contiguous arrays or `self.strides[0] == self.itemsize` for Fortran-style contiguous arrays is true.

Data in new `ndarrays` is in the row-major (C) order, unless otherwise specified, but, for example, *basic array slicing* often produces views in a different scheme.

**Note:** Several algorithms in NumPy work on arbitrarily strided arrays. However, some algorithms require single-segment arrays. When an irregularly strided array is passed in to such algorithms, a copy is automatically made.

## 1.1.4 Array attributes

Array attributes reflect information that is intrinsic to the array itself. Generally, accessing an array through its attributes allows you to get and sometimes set intrinsic properties of the array without creating a new array. The exposed attributes are the core parts of an array and only some of them can be reset meaningfully without creating a new array. Information on each attribute is given below.

### Memory layout

The following attributes contain information about the memory layout of the array:

| | |
|---|---|
| `ndarray.flags` | Information about the memory layout of the array. |
| `ndarray.shape` | Tuple of array dimensions. |
| `ndarray.strides` | Tuple of bytes to step in each dimension when traversing an array. |
| `ndarray.ndim` | Number of array dimensions. |
| `ndarray.data` | Python buffer object pointing to the start of the array's data. |
| `ndarray.size` | Number of elements in the array. |
| `ndarray.itemsize` | Length of one array element in bytes. |
| `ndarray.nbytes` | Total bytes consumed by the elements of the array. |
| `ndarray.base` | Base object if memory is from some other object. |

attribute

`ndarray.`**`flags`**

Information about the memory layout of the array.

### Notes

The *`flags`* object can be accessed dictionary-like (as in `a.flags['WRITEABLE']`), or by using lowercased attribute names (as in `a.flags.writeable`). Short flag names are only supported in dictionary access.

Only the WRITEBACKIFCOPY, UPDATEIFCOPY, WRITEABLE, and ALIGNED flags can be changed by the user, via direct assignment to the attribute or dictionary entry, or by calling *`ndarray.setflags`*.

The array flags cannot be set arbitrarily:

- UPDATEIFCOPY can only be set `False`.
- WRITEBACKIFCOPY can only be set `False`.
- ALIGNED can only be set `True` if the data is truly aligned.
- WRITEABLE can only be set `True` if the array owns its own memory or the ultimate owner of the memory exposes a writeable buffer interface or is a string.

Arrays can be both C-style and Fortran-style contiguous simultaneously. This is clear for 1-dimensional arrays, but can also be true for higher dimensional arrays.

Even for contiguous arrays a stride for a given dimension `arr.strides[dim]` may be *arbitrary* if `arr.shape[dim] == 1` or the array has no elements. It does *not* generally hold that `self.strides[-1] == self.itemsize` for C-style contiguous arrays or `self.strides[0] == self.itemsize` for Fortran-style contiguous arrays is true.

**Attributes**

**C_CONTIGUOUS (C)**

The data is in a single, C-style contiguous segment.

**F_CONTIGUOUS (F)**

The data is in a single, Fortran-style contiguous segment.

**OWNDATA (O)**

The array owns the memory it uses or borrows it from another object.

**WRITEABLE (W)**

The data area can be written to. Setting this to False locks the data, making it read-only. A view (slice, etc.) inherits WRITEABLE from its base array at creation time, but a view of a writeable array may be subsequently locked while the base array remains writeable. (The opposite is not true, in that a view of a locked array may not be made writeable. However, currently, locking a base object does not lock any views that already reference it, so under that circumstance it is possible to alter the contents of a locked array via a previously created writeable view onto it.) Attempting to change a non-writeable array raises a RuntimeError exception.

**ALIGNED (A)**

The data and all elements are aligned appropriately for the hardware.

**WRITEBACKIFCOPY (X)**

This array is a copy of some other array. The C-API function PyArray_ResolveWritebackIfCopy must be called before deallocating to the base array will be updated with the contents of this array.

**UPDATEIFCOPY (U)**

(Deprecated, use WRITEBACKIFCOPY) This array is a copy of some other array. When this array is deallocated, the base array will be updated with the contents of this array.

**FNC**

F_CONTIGUOUS and not C_CONTIGUOUS.

**FORC**

F_CONTIGUOUS or C_CONTIGUOUS (one-segment test).

**BEHAVED (B)**

ALIGNED and WRITEABLE.

**CARRAY (CA)**

BEHAVED and C_CONTIGUOUS.

**FARRAY (FA)**

BEHAVED and F_CONTIGUOUS and not C_CONTIGUOUS.

attribute

`ndarray.`**`shape`**

Tuple of array dimensions.

The shape property is usually used to get the current shape of an array, but may also be used to reshape the array in-place by assigning a tuple of array dimensions to it. As with *`numpy.reshape`*, one of the new shape dimensions can be -1, in which case its value is inferred from the size of the array and the remaining dimensions. Reshaping an array in-place will fail if a copy is required.

**See also:**

***`numpy.reshape`***

similar function

***`ndarray.reshape`***

similar method

### Examples

```
>>> x = np.array([1, 2, 3, 4])
>>> x.shape
(4,)
>>> y = np.zeros((2, 3, 4))
>>> y.shape
(2, 3, 4)
>>> y.shape = (3, 8)
>>> y
array([[ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.]])
>>> y.shape = (3, 6)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: total size of new array must be unchanged
>>> np.zeros((4,2))[::2].shape = (-1,)
```

(continues on next page)

(continued from previous page)

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: Incompatible shape for in-place modification. Use
`.reshape()` to make a copy with the desired shape.
```

attribute

`ndarray.`**`strides`**

Tuple of bytes to step in each dimension when traversing an array.

The byte offset of element `(i[0], i[1], ..., i[n])` in an array *a* is:

```
offset = sum(np.array(i) * a.strides)
```

A more detailed explanation of strides can be found in the "ndarray.rst" file in the NumPy reference guide.

**See also:**

*`numpy.lib.stride_tricks.as_strided`*

### Notes

Imagine an array of 32-bit integers (each 4 bytes):

```
x = np.array([[0, 1, 2, 3, 4],
              [5, 6, 7, 8, 9]], dtype=np.int32)
```

This array is stored in memory as 40 bytes, one after the other (known as a contiguous block of memory). The strides of an array tell us how many bytes we have to skip in memory to move to the next position along a certain axis. For example, we have to skip 4 bytes (1 value) to move to the next column, but 20 bytes (5 values) to get to the same position in the next row. As such, the strides for the array *x* will be `(20, 4)`.

### Examples

```
>>> y = np.reshape(np.arange(2*3*4), (2,3,4))
>>> y
array([[[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [ 8,  9, 10, 11]],
       [[12, 13, 14, 15],
        [16, 17, 18, 19],
        [20, 21, 22, 23]]])
>>> y.strides
(48, 16, 4)
>>> y[1,1,1]
17
>>> offset=sum(y.strides * np.array((1,1,1)))
>>> offset/y.itemsize
17
```

```
>>> x = np.reshape(np.arange(5*6*7*8), (5,6,7,8)).transpose(2,3,1,0)
>>> x.strides
(32, 4, 224, 1344)
>>> i = np.array([3,5,2,2])
```

(continues on next page)

(continued from previous page)

```
>>> offset = sum(i * x.strides)
>>> x[3,5,2,2]
813
>>> offset / x.itemsize
813
```

attribute

`ndarray.`**`ndim`**

Number of array dimensions.

### Examples

```
>>> x = np.array([1, 2, 3])
>>> x.ndim
1
>>> y = np.zeros((2, 3, 4))
>>> y.ndim
3
```

attribute

`ndarray.`**`data`**

Python buffer object pointing to the start of the array's data.

attribute

`ndarray.`**`size`**

Number of elements in the array.

Equal to `np.prod(a.shape)`, i.e., the product of the array's dimensions.

### Notes

*a.size* returns a standard arbitrary precision Python integer. This may not be the case with other methods of obtaining the same value (like the suggested `np.prod(a.shape)`, which returns an instance of `np.int_`), and may be relevant if the value is used further in calculations that may overflow a fixed size integer type.

### Examples

```
>>> x = np.zeros((3, 5, 2), dtype=np.complex128)
>>> x.size
30
>>> np.prod(x.shape)
30
```

attribute

`ndarray.`**`itemsize`**

Length of one array element in bytes.

### Examples

```
>>> x = np.array([1,2,3], dtype=np.float64)
>>> x.itemsize
8
>>> x = np.array([1,2,3], dtype=np.complex128)
>>> x.itemsize
16
```

attribute

`ndarray.`**`nbytes`**

Total bytes consumed by the elements of the array.

### Notes

Does not include memory consumed by non-element attributes of the array object.

### Examples

```
>>> x = np.zeros((3,5,2), dtype=np.complex128)
>>> x.nbytes
480
>>> np.prod(x.shape) * x.itemsize
480
```

attribute

`ndarray.`**`base`**

Base object if memory is from some other object.

### Examples

The base of an array that owns its memory is None:

```
>>> x = np.array([1,2,3,4])
>>> x.base is None
True
```

Slicing creates a view, whose memory is shared with x:

```
>>> y = x[2:]
>>> y.base is x
True
```

### Data type

**See also:**

*Data type objects*

The data type object associated with the array can be found in the `dtype` attribute:

| | |
|---|---|
| `ndarray.dtype` | Data-type of the array's elements. |

attribute

`ndarray.`**`dtype`**

Data-type of the array's elements.

**Parameters**

**None**

**Returns**

**d**

[numpy dtype object]

**See also:**

`numpy.dtype`

#### Examples

```
>>> x
array([[0, 1],
       [2, 3]])
>>> x.dtype
dtype('int32')
>>> type(x.dtype)
<type 'numpy.dtype'>
```

### Other attributes

| | |
|---|---|
| `ndarray.T` | The transposed array. |
| `ndarray.real` | The real part of the array. |
| `ndarray.imag` | The imaginary part of the array. |
| `ndarray.flat` | A 1-D iterator over the array. |
| `ndarray.ctypes` | An object to simplify the interaction of the array with the ctypes module. |

attribute

`ndarray.`**`T`**

The transposed array.

Same as `self.transpose()`.

**See also:**

*transpose*

### Examples

```
>>> x = np.array([[1.,2.],[3.,4.]])
>>> x
array([[ 1.,  2.],
       [ 3.,  4.]])
>>> x.T
array([[ 1.,  3.],
       [ 2.,  4.]])
>>> x = np.array([1.,2.,3.,4.])
>>> x
array([ 1.,  2.,  3.,  4.])
>>> x.T
array([ 1.,  2.,  3.,  4.])
```

attribute

`ndarray.`**`real`**

The real part of the array.

**See also:**

***numpy.real***

equivalent function

### Examples

```
>>> x = np.sqrt([1+0j, 0+1j])
>>> x.real
array([ 1.        ,  0.70710678])
>>> x.real.dtype
dtype('float64')
```

attribute

`ndarray.`**`imag`**

The imaginary part of the array.

### Examples

```
>>> x = np.sqrt([1+0j, 0+1j])
>>> x.imag
array([ 0.        ,  0.70710678])
>>> x.imag.dtype
dtype('float64')
```

attribute

`ndarray.`**`flat`**

A 1-D iterator over the array.

This is a *numpy.flatiter* instance, which acts similarly to, but is not a subclass of, Python's built-in iterator object.

**See also:**

***flatten***

Return a copy of the array collapsed into one dimension.

*flatiter*

### Examples

```
>>> x = np.arange(1, 7).reshape(2, 3)
>>> x
array([[1, 2, 3],
       [4, 5, 6]])
>>> x.flat[3]
4
>>> x.T
array([[1, 4],
       [2, 5],
       [3, 6]])
>>> x.T.flat[3]
5
>>> type(x.flat)
<class 'numpy.flatiter'>
```

An assignment example:

```
>>> x.flat = 3; x
array([[3, 3, 3],
       [3, 3, 3]])
>>> x.flat[[1,4]] = 1; x
array([[3, 1, 3],
       [3, 1, 3]])
```

attribute

`ndarray.`**`ctypes`**

An object to simplify the interaction of the array with the ctypes module.

This attribute creates an object that makes it easier to use arrays when calling shared libraries with the ctypes module. The returned object has, among others, data, shape, and strides attributes (see Notes below) which themselves return ctypes objects that can be used as arguments to a shared library.

**Parameters**

**None**

**Returns**

**c**

[Python object] Possessing attributes data, shape, strides, etc.

**See also:**

*`numpy.ctypeslib`*

## Notes

Below are the public attributes of this object which were documented in "Guide to NumPy" (we have omitted undocumented public attributes, as well as documented private attributes):

`_ctypes.`**`data`**

A pointer to the memory area of the array as a Python integer. This memory area may contain data that is not aligned, or not in correct byte-order. The memory area may not even be writeable. The array flags and data-type of this array should be respected when passing this attribute to arbitrary C-code to avoid trouble that can include Python crashing. User Beware! The value of this attribute is exactly the same as `self._array_interface_['data'][0]`.

Note that unlike `data_as`, a reference will not be kept to the array: code like `ctypes.c_void_p((a + b).ctypes.data)` will result in a pointer to a deallocated array, and should be spelt `(a + b).ctypes.data_as(ctypes.c_void_p)`

`_ctypes.`**`shape`**

(c_intp*self.ndim): A ctypes array of length self.ndim where the basetype is the C-integer corresponding to `dtype('p')` on this platform. This base-type could be `ctypes.c_int`, `ctypes.c_long`, or `ctypes.c_longlong` depending on the platform. The c_intp type is defined accordingly in *`numpy.ctypeslib`*. The ctypes array contains the shape of the underlying array.

`_ctypes.`**`strides`**

(c_intp*self.ndim): A ctypes array of length self.ndim where the basetype is the same as for the shape attribute. This ctypes array contains the strides information from the underlying array. This strides information is important for showing how many bytes must be jumped to get to the next element in the array.

`_ctypes.`**`data_as`**(*self*, *obj*)

Return the data pointer cast to a particular c-types object. For example, calling `self._as_parameter_` is equivalent to `self.data_as(ctypes.c_void_p)`. Perhaps you want to use the data as a pointer to a ctypes array of floating-point data: `self.data_as(ctypes.POINTER(ctypes.c_double))`.

The returned pointer will keep a reference to the array.

`_ctypes.`**`shape_as`**(*self*, *obj*)

Return the shape tuple as an array of some other c-types type. For example: `self.shape_as(ctypes.c_short)`.

`_ctypes.`**`strides_as`**(*self*, *obj*)

Return the strides tuple as an array of some other c-types type. For example: `self.strides_as(ctypes.c_longlong)`.

If the ctypes module is not available, then the ctypes attribute of array objects still returns something useful, but ctypes objects are not returned and errors may be raised instead. In particular, the object will still have the `as_parameter` attribute which will return an integer equal to the data attribute.

### Examples

```
>>> import ctypes
>>> x = np.array([[0, 1], [2, 3]], dtype=np.int32)
>>> x
array([[0, 1],
       [2, 3]], dtype=int32)
>>> x.ctypes.data
31962608 # may vary
>>> x.ctypes.data_as(ctypes.POINTER(ctypes.c_uint32))
<__main__.LP_c_uint object at 0x7ff2fc1fc200> # may vary
>>> x.ctypes.data_as(ctypes.POINTER(ctypes.c_uint32)).contents
c_uint(0)
>>> x.ctypes.data_as(ctypes.POINTER(ctypes.c_uint64)).contents
c_ulong(4294967296)
>>> x.ctypes.shape
<numpy.core._internal.c_long_Array_2 object at 0x7ff2fc1fce60> # may vary
>>> x.ctypes.strides
<numpy.core._internal.c_long_Array_2 object at 0x7ff2fc1ff320> # may vary
```

### Array interface

**See also:**

*The Array Interface*.

| | |
|---|---|
| *`__array_interface__`* | Python-side of the array interface |
| `__array_struct__` | C-side of the array interface |

### `ctypes` foreign function interface

| | |
|---|---|
| *`ndarray.ctypes`* | An object to simplify the interaction of the array with the ctypes module. |

## 1.1.5 Array methods

An *`ndarray`* object has many methods which operate on or with the array in some fashion, typically returning an array result. These methods are briefly explained below. (Each method's docstring has a more complete description.)

For the following methods there are also corresponding functions in *`numpy`*: *`all`*, *`any`*, *`argmax`*, *`argmin`*, *`argpartition`*, *`argsort`*, *`choose`*, *`clip`*, *`compress`*, *`copy`*, *`cumprod`*, *`cumsum`*, *`diagonal`*, *`imag`*, *`max`*, *`mean`*, *`min`*, *`nonzero`*, *`partition`*, *`prod`*, *`ptp`*, *`put`*, *`ravel`*, *`real`*, *`repeat`*, *`reshape`*, *`round`*, *`searchsorted`*, *`sort`*, *`squeeze`*, *`std`*, *`sum`*, *`swapaxes`*, *`take`*, *`trace`*, *`transpose`*, *`var`*.

### Array conversion

| | |
|---|---|
| *`ndarray.item`*(*args) | Copy an element of an array to a standard Python scalar and return it. |
| *`ndarray.tolist`*() | Return the array as an `a.ndim`-levels deep nested list of Python scalars. |
| *`ndarray.itemset`*(*args) | Insert scalar into an array (scalar is cast to array's dtype, if possible) |
| *`ndarray.tostring`*([order]) | A compatibility alias for *tobytes*, with exactly the same behavior. |
| *`ndarray.tobytes`*([order]) | Construct Python bytes containing the raw data bytes in the array. |
| *`ndarray.tofile`*(fid[, sep, format]) | Write array to a file as text or binary (default). |
| *`ndarray.dump`*(file) | Dump a pickle of the array to the specified file. |
| *`ndarray.dumps`*() | Returns the pickle of the array as a string. |
| *`ndarray.astype`*(dtype[, order, casting, …]) | Copy of the array, cast to a specified type. |
| *`ndarray.byteswap`*([inplace]) | Swap the bytes of the array elements |
| *`ndarray.copy`*([order]) | Return a copy of the array. |
| *`ndarray.view`*([dtype][, type]) | New view of array with the same data. |
| *`ndarray.getfield`*(dtype[, offset]) | Returns a field of the given array as a certain type. |
| *`ndarray.setflags`*([write, align, uic]) | Set array flags WRITEABLE, ALIGNED, (WRITEBACKIFCOPY and UPDATEIFCOPY), respectively. |
| *`ndarray.fill`*(value) | Fill the array with a scalar value. |

### Shape manipulation

For reshape, resize, and transpose, the single tuple argument may be replaced with `n` integers which will be interpreted as an n-tuple.

| | |
|---|---|
| *`ndarray.reshape`*(shape[, order]) | Returns an array containing the same data with a new shape. |
| *`ndarray.resize`*(new_shape[, refcheck]) | Change shape and size of array in-place. |
| *`ndarray.transpose`*(*axes) | Returns a view of the array with axes transposed. |
| *`ndarray.swapaxes`*(axis1, axis2) | Return a view of the array with *axis1* and *axis2* interchanged. |
| *`ndarray.flatten`*([order]) | Return a copy of the array collapsed into one dimension. |
| *`ndarray.ravel`*([order]) | Return a flattened array. |
| *`ndarray.squeeze`*([axis]) | Remove single-dimensional entries from the shape of *a*. |

### Item selection and manipulation

For array methods that take an *axis* keyword, it defaults to *None*. If axis is *None*, then the array is treated as a 1-D array. Any other value for *axis* represents the dimension along which the operation should proceed.

| | |
|---|---|
| *`ndarray.take`*(indices[, axis, out, mode]) | Return an array formed from the elements of *a* at the given indices. |
| *`ndarray.put`*(indices, values[, mode]) | Set `a.flat[n] = values[n]` for all *n* in indices. |
| *`ndarray.repeat`*(repeats[, axis]) | Repeat elements of an array. |

Continued on next page

Table 9 – continued from previous page

| | |
|---|---|
| `ndarray.choose`(choices[, out, mode]) | Use an index array to construct a new array from a set of choices. |
| `ndarray.sort`([axis, kind, order]) | Sort an array in-place. |
| `ndarray.argsort`([axis, kind, order]) | Returns the indices that would sort this array. |
| `ndarray.partition`(kth[, axis, kind, order]) | Rearranges the elements in the array in such a way that the value of the element in kth position is in the position it would be in a sorted array. |
| `ndarray.argpartition`(kth[, axis, kind, order]) | Returns the indices that would partition this array. |
| `ndarray.searchsorted`(v[, side, sorter]) | Find indices where elements of v should be inserted in a to maintain order. |
| `ndarray.nonzero`() | Return the indices of the elements that are non-zero. |
| `ndarray.compress`(condition[, axis, out]) | Return selected slices of this array along given axis. |
| `ndarray.diagonal`([offset, axis1, axis2]) | Return specified diagonals. |

### Calculation

Many of these methods take an argument named *axis*. In such cases,

- If *axis* is *None* (the default), the array is treated as a 1-D array and the operation is performed over the entire array. This behavior is also the default if self is a 0-dimensional array or array scalar. (An array scalar is an instance of the types/classes float32, float64, etc., whereas a 0-dimensional array is an ndarray instance containing precisely one array scalar.)
- If *axis* is an integer, then the operation is done over the given axis (for each 1-D subarray that can be created along the given axis).

---

**Example of the *axis* argument**

A 3-dimensional array of size 3 x 3 x 3, summed over each of its three axes

```
>>> x = np.arange(27).reshape((3,3,3))
>>> x
array([[[ 0,  1,  2],
        [ 3,  4,  5],
        [ 6,  7,  8]],
       [[ 9, 10, 11],
        [12, 13, 14],
        [15, 16, 17]],
       [[18, 19, 20],
        [21, 22, 23],
        [24, 25, 26]]])
>>> x.sum(axis=0)
array([[27, 30, 33],
       [36, 39, 42],
       [45, 48, 51]])
>>> # for sum, axis is the first keyword, so we may omit it,
>>> # specifying only its value
>>> x.sum(0), x.sum(1), x.sum(2)
(array([[27, 30, 33],
        [36, 39, 42],
        [45, 48, 51]]),
 array([[ 9, 12, 15],
        [36, 39, 42],
        [63, 66, 69]]),
```

(continues on next page)

(continued from previous page)

```
array([[ 3, 12, 21],
       [30, 39, 48],
       [57, 66, 75]]))
```

The parameter *dtype* specifies the data type over which a reduction operation (like summing) should take place. The default reduce data type is the same as the data type of *self*. To avoid overflow, it can be useful to perform the reduction using a larger data type.

For several methods, an optional *out* argument can also be provided and the result will be placed into the output array given. The *out* argument must be an `ndarray` and have the same number of elements. It can have a different data type in which case casting will be performed.

| | |
|---|---|
| `ndarray.max`([axis, out, keepdims, initial, …]) | Return the maximum along a given axis. |
| `ndarray.argmax`([axis, out]) | Return indices of the maximum values along the given axis. |
| `ndarray.min`([axis, out, keepdims, initial, …]) | Return the minimum along a given axis. |
| `ndarray.argmin`([axis, out]) | Return indices of the minimum values along the given axis of *a*. |
| `ndarray.ptp`([axis, out, keepdims]) | Peak to peak (maximum - minimum) value along a given axis. |
| `ndarray.clip`([min, max, out]) | Return an array whose values are limited to `[min, max]`. |
| `ndarray.conj`() | Complex-conjugate all elements. |
| `ndarray.round`([decimals, out]) | Return *a* with each element rounded to the given number of decimals. |
| `ndarray.trace`([offset, axis1, axis2, dtype, out]) | Return the sum along diagonals of the array. |
| `ndarray.sum`([axis, dtype, out, keepdims, …]) | Return the sum of the array elements over the given axis. |
| `ndarray.cumsum`([axis, dtype, out]) | Return the cumulative sum of the elements along the given axis. |
| `ndarray.mean`([axis, dtype, out, keepdims]) | Returns the average of the array elements along given axis. |
| `ndarray.var`([axis, dtype, out, ddof, keepdims]) | Returns the variance of the array elements, along given axis. |
| `ndarray.std`([axis, dtype, out, ddof, keepdims]) | Returns the standard deviation of the array elements along given axis. |
| `ndarray.prod`([axis, dtype, out, keepdims, …]) | Return the product of the array elements over the given axis |
| `ndarray.cumprod`([axis, dtype, out]) | Return the cumulative product of the elements along the given axis. |
| `ndarray.all`([axis, out, keepdims]) | Returns True if all elements evaluate to True. |
| `ndarray.any`([axis, out, keepdims]) | Returns True if any of the elements of *a* evaluate to True. |

### 1.1.6 Arithmetic, matrix multiplication, and comparison operations

Arithmetic and comparison operations on *ndarrays* are defined as element-wise operations, and generally yield *ndarray* objects as results.

Each of the arithmetic operations (`+`, `-`, `*`, `/`, `//`, `%`, `divmod()`, `**` or `pow()`, `<<`, `>>`, `&`, `^`, `|`, `~`) and the comparisons (`==`, `<`, `>`, `<=`, `>=`, `!=`) is equivalent to the corresponding universal function (or ufunc for short) in NumPy. For more information, see the section on *Universal Functions*.

Comparison operators:

| | |
|---|---|
| `ndarray.__lt__`(self, value, /) | Return self<value. |
| `ndarray.__le__`(self, value, /) | Return self<=value. |
| `ndarray.__gt__`(self, value, /) | Return self>value. |
| `ndarray.__ge__`(self, value, /) | Return self>=value. |
| `ndarray.__eq__`(self, value, /) | Return self==value. |
| `ndarray.__ne__`(self, value, /) | Return self!=value. |

method

`ndarray.`**`__lt__`**(*self*, *value*, /)
Return self<value.

method

`ndarray.`**`__le__`**(*self*, *value*, /)
Return self<=value.

method

`ndarray.`**`__gt__`**(*self*, *value*, /)
Return self>value.

method

`ndarray.`**`__ge__`**(*self*, *value*, /)
Return self>=value.

method

`ndarray.`**`__eq__`**(*self*, *value*, /)
Return self==value.

method

`ndarray.`**`__ne__`**(*self*, *value*, /)
Return self!=value.

Truth value of an array (`bool`):

| | |
|---|---|
| `ndarray.__bool__`(self, /) | self != 0 |

method

`ndarray.`**`__bool__`**(*self*, /)
self != 0

**Note:** Truth-value testing of an array invokes `ndarray.__bool__`, which raises an error if the number of elements in the array is larger than 1, because the truth value of such arrays is ambiguous. Use `.any()` and `.all()` instead to be clear about what is meant in such cases. (If the number of elements is 0, the array evaluates to `False`.)

---

Unary operations:

| | |
|---|---|
| *ndarray.__neg__*(self, /) | -self |
| *ndarray.__pos__*(self, /) | +self |
| *ndarray.__abs__*(self) | |
| *ndarray.__invert__*(self, /) | ~self |

method

`ndarray.`**`__neg__`**(*self*, */*)
-self

method

`ndarray.`**`__pos__`**(*self*, */*)
+self

method

`ndarray.`**`__abs__`**(*self*)

method

`ndarray.`**`__invert__`**(*self*, */*)
~self

Arithmetic:

| | |
|---|---|
| *ndarray.__add__*(self, value, /) | Return self+value. |
| *ndarray.__sub__*(self, value, /) | Return self-value. |
| *ndarray.__mul__*(self, value, /) | Return self*value. |
| *ndarray.__truediv__*(self, value, /) | Return self/value. |
| *ndarray.__floordiv__*(self, value, /) | Return self//value. |
| *ndarray.__mod__*(self, value, /) | Return self%value. |
| *ndarray.__divmod__*(self, value, /) | Return divmod(self, value). |
| *ndarray.__pow__*(self, value[, mod]) | Return pow(self, value, mod). |
| *ndarray.__lshift__*(self, value, /) | Return self«value. |
| *ndarray.__rshift__*(self, value, /) | Return self»value. |
| *ndarray.__and__*(self, value, /) | Return self&value. |
| *ndarray.__or__*(self, value, /) | Return self\|value. |
| *ndarray.__xor__*(self, value, /) | Return self^value. |

method

`ndarray.`**`__add__`**(*self*, *value*, */*)
Return self+value.

method

`ndarray.`**`__sub__`**(*self*, *value*, */*)
Return self-value.

method

`ndarray.`**`__mul__`**(*self*, *value*, */*)
Return self*value.

method

`ndarray.`**`__truediv__`**(*self*, *value*, /)
Return self/value.

method

`ndarray.`**`__floordiv__`**(*self*, *value*, /)
Return self//value.

method

`ndarray.`**`__mod__`**(*self*, *value*, /)
Return self%value.

method

`ndarray.`**`__divmod__`**(*self*, *value*, /)
Return divmod(self, value).

method

`ndarray.`**`__pow__`**(*self*, *value*, *mod=None*, /)
Return pow(self, value, mod).

method

`ndarray.`**`__lshift__`**(*self*, *value*, /)
Return self«value.

method

`ndarray.`**`__rshift__`**(*self*, *value*, /)
Return self»value.

method

`ndarray.`**`__and__`**(*self*, *value*, /)
Return self&value.

method

`ndarray.`**`__or__`**(*self*, *value*, /)
Return self|value.

method

`ndarray.`**`__xor__`**(*self*, *value*, /)
Return self^value.

**Note:**

- Any third argument to `pow` is silently ignored, as the underlying `ufunc` takes only two arguments.
- Because `ndarray` is a built-in type (written in C), the `__r{op}__` special methods are not directly defined.
- The functions called to implement many arithmetic special methods for arrays can be modified using `__array_ufunc__`.

Arithmetic, in-place:

| | |
|---|---|
| `ndarray.__iadd__`(self, value, /) | Return self+=value. |
| `ndarray.__isub__`(self, value, /) | Return self-=value. |

Continued on next page

Table 15 – continued from previous page

| | |
|---|---|
| `ndarray.__imul__`(self, value, /) | Return self*=value. |
| `ndarray.__itruediv__`(self, value, /) | Return self/=value. |
| `ndarray.__ifloordiv__`(self, value, /) | Return self//=value. |
| `ndarray.__imod__`(self, value, /) | Return self%=value. |
| `ndarray.__ipow__`(self, value, /) | Return self**=value. |
| `ndarray.__ilshift__`(self, value, /) | Return self«=value. |
| `ndarray.__irshift__`(self, value, /) | Return self»=value. |
| `ndarray.__iand__`(self, value, /) | Return self&=value. |
| `ndarray.__ior__`(self, value, /) | Return self\|=value. |
| `ndarray.__ixor__`(self, value, /) | Return self^=value. |

method

`ndarray.`**`__iadd__`**(*self*, *value*, /)
Return self+=value.

method

`ndarray.`**`__isub__`**(*self*, *value*, /)
Return self-=value.

method

`ndarray.`**`__imul__`**(*self*, *value*, /)
Return self*=value.

method

`ndarray.`**`__itruediv__`**(*self*, *value*, /)
Return self/=value.

method

`ndarray.`**`__ifloordiv__`**(*self*, *value*, /)
Return self//=value.

method

`ndarray.`**`__imod__`**(*self*, *value*, /)
Return self%=value.

method

`ndarray.`**`__ipow__`**(*self*, *value*, /)
Return self**=value.

method

`ndarray.`**`__ilshift__`**(*self*, *value*, /)
Return self«=value.

method

`ndarray.`**`__irshift__`**(*self*, *value*, /)
Return self»=value.

method

`ndarray.`**`__iand__`**(*self*, *value*, /)
Return self&=value.

method

`ndarray.`**`__ior__`**`(`*self*, *value*, `/`)

Return self|=value.

method

`ndarray.`**`__ixor__`**`(`*self*, *value*, `/`)

Return self^=value.

**Warning:** In place operations will perform the calculation using the precision decided by the data type of the two operands, but will silently downcast the result (if necessary) so it can fit back into the array. Therefore, for mixed precision calculations, `A {op}= B` can be different than `A = A {op} B`. For example, suppose `a = ones((3, 3))`. Then, `a += 3j` is different than `a = a + 3j`: while they both perform the same computation, `a += 3` casts the result to fit back in `a`, whereas `a = a + 3j` re-binds the name `a` to the result.

Matrix Multiplication:

| | |
|---|---|
| *`ndarray.__matmul__`*(self, value, /) | Return self@value. |

method

`ndarray.`**`__matmul__`**`(`*self*, *value*, `/`)

Return self@value.

**Note:** Matrix operators `@` and `@=` were introduced in Python 3.5 following PEP465. NumPy 1.10.0 has a preliminary implementation of `@` for testing purposes. Further documentation can be found in the *`matmul`* documentation.

## 1.1.7 Special methods

For standard library functions:

| | |
|---|---|
| *`ndarray.__copy__`*() | Used if `copy.copy` is called on an array. |
| *`ndarray.__deepcopy__`*() | Used if `copy.deepcopy` is called on an array. |
| *`ndarray.__reduce__`*() | For pickling. |
| *`ndarray.__setstate__`*(state, /) | For unpickling. |

method

`ndarray.`**`__copy__`**`()`

Used if `copy.copy` is called on an array. Returns a copy of the array.

Equivalent to `a.copy(order='K')`.

method

`ndarray.`**`__deepcopy__`**`()`

Used if `copy.deepcopy` is called on an array.

method

`ndarray.`**`__reduce__`**`()`

For pickling.

method

`ndarray.`**`__setstate__`**(*state*, /)

For unpickling.

The *state* argument must be a sequence that contains the following elements:

**Parameters**

**version**

[int] optional pickle version. If omitted defaults to 0.

**shape**

[tuple]

**dtype**

[data-type]

**isFortran**

[bool]

**rawdata**

[string or list] a binary string with the data (or a list if 'a' is an object array)

Basic customization:

| | |
|---|---|
| *`ndarray.__new__`*(*args, **kwargs) | Create and return a new object. |
| *`ndarray.__array__`*() | Returns either a new reference to self if dtype is not given or a new array of provided data type if dtype is different from the current dtype of the array. |
| *`ndarray.__array_wrap__`*() | |

method

`ndarray.`**`__new__`**(**args*, ***kwargs*)

Create and return a new object. See help(type) for accurate signature.

method

`ndarray.`**`__array__`**()

Returns either a new reference to self if dtype is not given or a new array of provided data type if dtype is different from the current dtype of the array.

method

`ndarray.`**`__array_wrap__`**()

Container customization: (see *Indexing*)

| | |
|---|---|
| *`ndarray.__len__`*(self, /) | Return len(self). |
| *`ndarray.__getitem__`*(self, key, /) | Return self[key]. |
| *`ndarray.__setitem__`*(self, key, value, /) | Set self[key] to value. |
| *`ndarray.__contains__`*(self, key, /) | Return key in self. |

method

`ndarray.`**`__len__`**(*self*, /)

Return len(self).

method

`ndarray.`**`__getitem__`**`(`*self*, *key*, */*`)`
Return self[key].

method

`ndarray.`**`__setitem__`**`(`*self*, *key*, *value*, */*`)`
Set self[key] to value.

method

`ndarray.`**`__contains__`**`(`*self*, *key*, */*`)`
Return key in self.

Conversion; the operations `int`, `float` and `complex`. . They work only on arrays that have one element in them and return the appropriate scalar.

| | |
|---|---|
| *`ndarray.__int__`*(self) | |
| *`ndarray.__float__`*(self) | |
| *`ndarray.__complex__`*() | |

method

`ndarray.`**`__int__`**`(`*self*`)`

method

`ndarray.`**`__float__`**`(`*self*`)`

method

`ndarray.`**`__complex__`**`()`

String representations:

| | |
|---|---|
| *`ndarray.__str__`*(self, /) | Return str(self). |
| *`ndarray.__repr__`*(self, /) | Return repr(self). |

method

`ndarray.`**`__str__`**`(`*self*, */*`)`
Return str(self).

method

`ndarray.`**`__repr__`**`(`*self*, */*`)`
Return repr(self).

## 1.2 Scalars

Python defines only one type of a particular data class (there is only one integer type, one floating-point type, etc.). This can be convenient in applications that don't need to be concerned with all the ways data can be represented in a computer. For scientific computing, however, more control is often needed.

In NumPy, there are 24 new fundamental Python types to describe different types of scalars. These type descriptors are mostly based on the types available in the C language that CPython is written in, with several additional types compatible with Python's types.

Array scalars have the same attributes and methods as `ndarrays`.[1] This allows one to treat items of an array partly on the same footing as arrays, smoothing out rough edges that result when mixing scalar and array operations.

Array scalars live in a hierarchy (see the Figure below) of data types. They can be detected using the hierarchy: For example, `isinstance(val, np.generic)` will return `True` if *val* is an array scalar object. Alternatively, what kind of array scalar is present can be determined using other members of the data type hierarchy. Thus, for example `isinstance(val, np.complexfloating)` will return `True` if *val* is a complex valued type, while `isinstance(val, np.flexible)` will return true if *val* is one of the flexible itemsize array types (`string`, `unicode`, `void`).

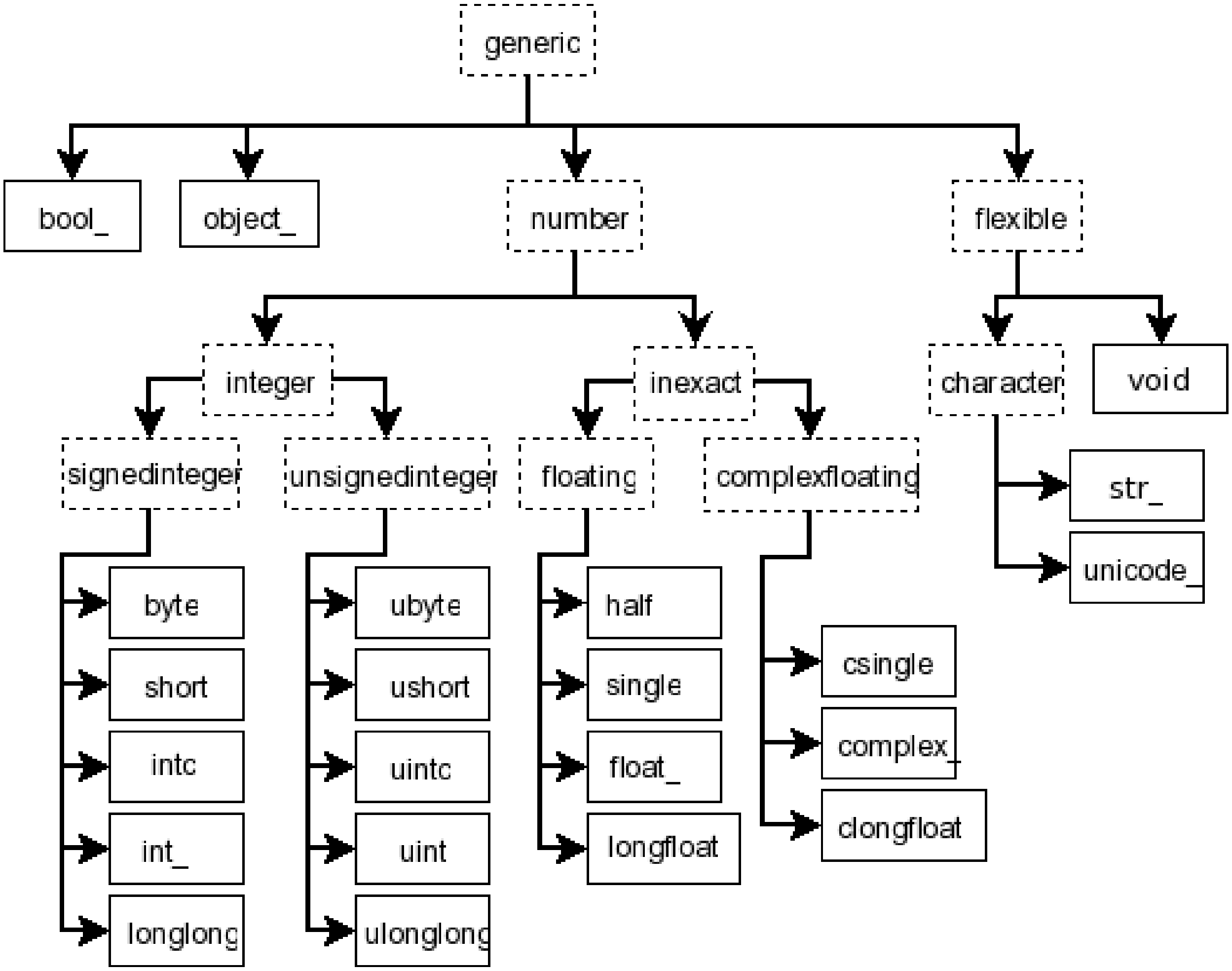

generic
bool_
object_
number
flexible
integer
inexact
character
void
signedinteger
unsignedinteger
floating
complexfloating
str_
unicode_
byte
short
intc
int_
longlong
ubyte
ushort
uintc
uint
ulonglong
half
single
float_
longfloat
csingle
complex_
clongfloat

Fig. 2: **Figure:** Hierarchy of type objects representing the array data types. Not shown are the two integer types `intp` and `uintp` which just point to the integer type that holds a pointer for the platform. All the number types can be obtained using bit-width names as well.

---

[1] However, array scalars are immutable, so none of the array scalar attributes are settable.

### 1.2.1 Built-in scalar types

The built-in scalar types are shown below. Along with their (mostly) C-derived names, the integer, float, and complex data-types are also available using a bit-width convention so that an array of the right size can always be ensured (e.g. `int8`, `float64`, `complex128`). Two aliases (`intp` and `uintp`) pointing to the integer type that is sufficiently large to hold a C pointer are also provided. The C-like names are associated with character codes, which are shown in the table. Use of the character codes, however, is discouraged.

Some of the scalar types are essentially equivalent to fundamental Python types and therefore inherit from them as well as from the generic array scalar type:

| Array scalar type | Related Python type |
|---|---|
| `int_` | `IntType` (Python 2 only) |
| `float_` | `FloatType` |
| `complex_` | `ComplexType` |
| `bytes_` | `BytesType` |
| `unicode_` | `UnicodeType` |

The `bool_` data type is very similar to the Python `BooleanType` but does not inherit from it because Python's `BooleanType` does not allow itself to be inherited from, and on the C-level the size of the actual bool data is not the same as a Python Boolean scalar.

**Warning:** The `bool_` type is not a subclass of the `int_` type (the `bool_` is not even a number type). This is different than Python's default implementation of `bool` as a sub-class of int.

**Warning:** The `int_` type does **not** inherit from the `int` built-in under Python 3, because type `int` is no longer a fixed-width integer type.

**Tip:** The default data type in NumPy is `float_`.

In the tables below, `platform?` means that the type may not be available on all platforms. Compatibility with different C or Python types is indicated: two types are compatible if their data is of the same size and interpreted in the same way.

Booleans:

| Type | Remarks | Character code |
|---|---|---|
| `bool_` | compatible: Python bool | `'?'` |
| `bool8` | 8 bits | |

Integers:

| | | |
|---|---|---|
| `byte` | compatible: C char | `'b'` |
| `short` | compatible: C short | `'h'` |
| `intc` | compatible: C int | `'i'` |
| `int_` | compatible: Python int | `'l'` |
| `longlong` | compatible: C long long | `'q'` |
| `intp` | large enough to fit a pointer | `'p'` |
| `int8` | 8 bits | |
| `int16` | 16 bits | |
| `int32` | 32 bits | |
| `int64` | 64 bits | |

Unsigned integers:

| | | |
|---|---|---|
| `ubyte` | compatible: C unsigned char | `'B'` |
| `ushort` | compatible: C unsigned short | `'H'` |
| `uintc` | compatible: C unsigned int | `'I'` |
| `uint` | compatible: Python int | `'L'` |
| `ulonglong` | compatible: C long long | `'Q'` |
| `uintp` | large enough to fit a pointer | `'P'` |
| `uint8` | 8 bits | |
| `uint16` | 16 bits | |
| `uint32` | 32 bits | |
| `uint64` | 64 bits | |

Floating-point numbers:

| | | |
|---|---|---|
| `half` | | `'e'` |
| `single` | compatible: C float | `'f'` |
| `double` | compatible: C double | |
| `float_` | compatible: Python float | `'d'` |
| `longfloat` | compatible: C long float | `'g'` |
| `float16` | 16 bits | |
| `float32` | 32 bits | |
| `float64` | 64 bits | |
| `float96` | 96 bits, platform? | |
| `float128` | 128 bits, platform? | |

Complex floating-point numbers:

| | | |
|---|---|---|
| `csingle` | | `'F'` |
| `complex_` | compatible: Python complex | `'D'` |
| `clongfloat` | | `'G'` |
| `complex64` | two 32-bit floats | |
| `complex128` | two 64-bit floats | |
| `complex192` | two 96-bit floats, platform? | |
| `complex256` | two 128-bit floats, platform? | |

Any Python object:

| | | |
|---|---|---|
| `object_` | any Python object | `'O'` |

**Note:** The data actually stored in object arrays (*i.e.*, arrays having dtype `object_`) are references to Python objects, not the objects themselves. Hence, object arrays behave more like usual Python `lists`, in the sense that their contents need not be of the same Python type.

The object type is also special because an array containing `object_` items does not return an `object_` object on item access, but instead returns the actual object that the array item refers to.

The following data types are **flexible**: they have no predefined size and the data they describe can be of different length in different arrays. (In the character codes `#` is an integer denoting how many elements the data type consists of.)

| | | |
|---|---|---|
| `bytes_` | compatible: Python bytes | `'S#'` |
| `unicode_` | compatible: Python unicode/str | `'U#'` |
| `void` | | `'V#'` |

**Warning:** See *Note on string types*.

Numeric Compatibility: If you used old typecode characters in your Numeric code (which was never recommended), you will need to change some of them to the new characters. In particular, the needed changes are `c -> S1`, `b -> B`, `1 -> b`, `s -> h`, `w -> H`, and `u -> I`. These changes make the type character convention more consistent with other Python modules such as the `struct` module.

## 1.2.2 Attributes

The array scalar objects have an `array priority` of *`NPY_SCALAR_PRIORITY`* (-1,000,000.0). They also do not (yet) have a *`ctypes`* attribute. Otherwise, they share the same attributes as arrays:

| | |
|---|---|
| *`generic.flags`* | integer value of flags |
| *`generic.shape`* | tuple of array dimensions |
| *`generic.strides`* | tuple of bytes steps in each dimension |
| *`generic.ndim`* | number of array dimensions |
| *`generic.data`* | pointer to start of data |
| *`generic.size`* | number of elements in the gentype |
| *`generic.itemsize`* | length of one element in bytes |
| *`generic.base`* | base object |
| *`generic.dtype`* | get array data-descriptor |
| *`generic.real`* | real part of scalar |
| *`generic.imag`* | imaginary part of scalar |
| *`generic.flat`* | a 1-d view of scalar |
| *`generic.T`* | transpose |
| *`generic.__array_interface__`* | Array protocol: Python side |
| *`generic.__array_struct__`* | Array protocol: struct |
| *`generic.__array_priority__`* | Array priority. |
| *`generic.__array_wrap__`*() | sc.__array_wrap__(obj) return scalar from array |

attribute

`generic.`**`flags`**

integer value of flags

attribute

`generic.`**`shape`**
tuple of array dimensions

attribute

`generic.`**`strides`**
tuple of bytes steps in each dimension

attribute

`generic.`**`ndim`**
number of array dimensions

attribute

`generic.`**`data`**
pointer to start of data

attribute

`generic.`**`size`**
number of elements in the gentype

attribute

`generic.`**`itemsize`**
length of one element in bytes

attribute

`generic.`**`base`**
base object

attribute

`generic.`**`dtype`**
get array data-descriptor

attribute

`generic.`**`real`**
real part of scalar

attribute

`generic.`**`imag`**
imaginary part of scalar

attribute

`generic.`**`flat`**
a 1-d view of scalar

attribute

`generic.`**`T`**
transpose

attribute

`generic.`**`__array_interface__`**
Array protocol: Python side

attribute

`generic.`**`__array_struct__`**
Array protocol: struct

attribute

`generic.`**`__array_priority__`**

Array priority.

method

`generic.`**`__array_wrap__`**`()`

sc.__array_wrap__(obj) return scalar from array

## 1.2.3 Indexing

**See also:**

*Indexing*, *Data type objects (dtype)*

Array scalars can be indexed like 0-dimensional arrays: if *x* is an array scalar,

- `x[()]` returns a copy of array scalar
- `x[...]` returns a 0-dimensional `ndarray`
- `x['field-name']` returns the array scalar in the field *field-name*. (*x* can have fields, for example, when it corresponds to a structured data type.)

## 1.2.4 Methods

Array scalars have exactly the same methods as arrays. The default behavior of these methods is to internally convert the scalar to an equivalent 0-dimensional array and to call the corresponding array method. In addition, math operations on array scalars are defined so that the same hardware flags are set and used to interpret the results as for *ufunc*, so that the error state used for ufuncs also carries over to the math on array scalars.

The exceptions to the above rules are given below:

| | |
|---|---|
| `generic` | Base class for numpy scalar types. |
| `generic.__array__()` | sc.__array__(dtype) return 0-dim array from scalar with specified dtype |
| `generic.__array_wrap__()` | sc.__array_wrap__(obj) return scalar from array |
| `generic.squeeze()` | Not implemented (virtual attribute) |
| `generic.byteswap()` | Not implemented (virtual attribute) |
| `generic.__reduce__()` | Helper for pickle. |
| `generic.__setstate__()` | |
| `generic.setflags()` | Not implemented (virtual attribute) |

**`class`** `numpy.`**`generic`**

Base class for numpy scalar types.

Class from which most (all?) numpy scalar types are derived. For consistency, exposes the same API as `ndarray`, despite many consequent attributes being either "get-only," or completely irrelevant. This is the class from which it is strongly suggested users should derive custom scalar types.

**Attributes**

***`T`***

transpose

***`base`***

base object

***`data`***

pointer to start of data

***`dtype`***

get array data-descriptor

***`flags`***

integer value of flags

***`flat`***

a 1-d view of scalar

***`imag`***

imaginary part of scalar

***`itemsize`***

length of one element in bytes

**`nbytes`**

length of item in bytes

***`ndim`***

number of array dimensions

***`real`***

real part of scalar

***`shape`***

tuple of array dimensions

***`size`***

number of elements in the gentype

***`strides`***

tuple of bytes steps in each dimension

### Methods

| | |
|---|---|
| *`all`*() | Not implemented (virtual attribute) |
| *`any`*() | Not implemented (virtual attribute) |
| *`argmax`*() | Not implemented (virtual attribute) |
| *`argmin`*() | Not implemented (virtual attribute) |
| *`argsort`*() | Not implemented (virtual attribute) |
| *`astype`*() | Not implemented (virtual attribute) |
| *`byteswap`*() | Not implemented (virtual attribute) |
| *`choose`*() | Not implemented (virtual attribute) |
| *`clip`*() | Not implemented (virtual attribute) |
| *`compress`*() | Not implemented (virtual attribute) |
| *`conjugate`*() | Not implemented (virtual attribute) |

Continued on next page

Table 24 – continued from previous page

| | |
|---|---|
| *copy*() | Not implemented (virtual attribute) |
| *cumprod*() | Not implemented (virtual attribute) |
| *cumsum*() | Not implemented (virtual attribute) |
| *diagonal*() | Not implemented (virtual attribute) |
| *dump*() | Not implemented (virtual attribute) |
| *dumps*() | Not implemented (virtual attribute) |
| *fill*() | Not implemented (virtual attribute) |
| *flatten*() | Not implemented (virtual attribute) |
| *getfield*() | Not implemented (virtual attribute) |
| *item*() | Not implemented (virtual attribute) |
| *itemset*() | Not implemented (virtual attribute) |
| *max*() | Not implemented (virtual attribute) |
| *mean*() | Not implemented (virtual attribute) |
| *min*() | Not implemented (virtual attribute) |
| *newbyteorder*([new_order]) | Return a new *dtype* with a different byte order. |
| *nonzero*() | Not implemented (virtual attribute) |
| *prod*() | Not implemented (virtual attribute) |
| *ptp*() | Not implemented (virtual attribute) |
| *put*() | Not implemented (virtual attribute) |
| *ravel*() | Not implemented (virtual attribute) |
| *repeat*() | Not implemented (virtual attribute) |
| *reshape*() | Not implemented (virtual attribute) |
| *resize*() | Not implemented (virtual attribute) |
| *round*() | Not implemented (virtual attribute) |
| *searchsorted*() | Not implemented (virtual attribute) |
| *setfield*() | Not implemented (virtual attribute) |
| *setflags*() | Not implemented (virtual attribute) |
| *sort*() | Not implemented (virtual attribute) |
| *squeeze*() | Not implemented (virtual attribute) |
| *std*() | Not implemented (virtual attribute) |
| *sum*() | Not implemented (virtual attribute) |
| *swapaxes*() | Not implemented (virtual attribute) |
| *take*() | Not implemented (virtual attribute) |
| *tofile*() | Not implemented (virtual attribute) |
| *tolist*() | Not implemented (virtual attribute) |
| *tostring*() | Not implemented (virtual attribute) |
| *trace*() | Not implemented (virtual attribute) |
| *transpose*() | Not implemented (virtual attribute) |
| *var*() | Not implemented (virtual attribute) |
| *view*() | Not implemented (virtual attribute) |

method

`generic.`**`all`**`()`

Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also the corresponding attribute of the derived class of interest.

method

`generic.`**`any`**`()`

Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also the corresponding attribute of the derived class of interest.

method

`generic.`**`argmax`**`()`

Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also the corresponding attribute of the derived class of interest.

method

`generic.`**`argmin`**`()`

Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also the corresponding attribute of the derived class of interest.

method

`generic.`**`argsort`**`()`

Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also the corresponding attribute of the derived class of interest.

method

`generic.`**`astype`**`()`

Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also the corresponding attribute of the derived class of interest.

method

`generic.`**`byteswap`**`()`

Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also the corresponding attribute of the derived class of interest.

method

`generic.`**`choose`**`()`

Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also the corresponding attribute of the derived class of interest.

method

`generic.clip()`

Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also the corresponding attribute of the derived class of interest.

method

`generic.compress()`

Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also the corresponding attribute of the derived class of interest.

method

`generic.conjugate()`

Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also the corresponding attribute of the derived class of interest.

method

`generic.copy()`

Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also the corresponding attribute of the derived class of interest.

method

`generic.cumprod()`

Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also the corresponding attribute of the derived class of interest.

method

`generic.cumsum()`

Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also the corresponding attribute of the derived class of interest.

method

`generic.diagonal()`

Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also the corresponding attribute of the derived class of interest.

method

`generic.`**`dump`**`()`

Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also the corresponding attribute of the derived class of interest.

method

`generic.`**`dumps`**`()`

Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also the corresponding attribute of the derived class of interest.

method

`generic.`**`fill`**`()`

Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also the corresponding attribute of the derived class of interest.

method

`generic.`**`flatten`**`()`

Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also the corresponding attribute of the derived class of interest.

method

`generic.`**`getfield`**`()`

Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also the corresponding attribute of the derived class of interest.

method

`generic.`**`item`**`()`

Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also the corresponding attribute of the derived class of interest.

method

`generic.`**`itemset`**`()`

Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also the corresponding attribute of the derived class of interest.

method

`generic.`**`max`**`()`

Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also the corresponding attribute of the derived class of interest.

method

`generic.`**`mean`**`()`

Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also the corresponding attribute of the derived class of interest.

method

`generic.`**`min`**`()`

Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also the corresponding attribute of the derived class of interest.

method

`generic.`**`newbyteorder`**`(`*new_order='S'*`)`

Return a new `dtype` with a different byte order.

Changes are also made in all fields and sub-arrays of the data type.

The *new_order* code can be any from the following:

- 'S' - swap dtype from current to opposite endian
- {'<', 'L'} - little endian
- {'>', 'B'} - big endian
- {'=', 'N'} - native order
- {'|', 'I'} - ignore (no change to byte order)

**Parameters**

**new_order**

[str, optional] Byte order to force; a value from the byte order specifications above. The default value ('S') results in swapping the current byte order. The code does a case-insensitive check on the first letter of *new_order* for the alternatives above. For example, any of 'B' or 'b' or 'biggish' are valid to specify big-endian.

**Returns**

**new_dtype**

[dtype] New `dtype` object with the given change to the byte order.

method

`generic.`**`nonzero`**`()`

Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also the corresponding attribute of the derived class of interest.

method

`generic.`**`prod`**`()`

Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also the corresponding attribute of the derived class of interest.

method

`generic.`**`ptp`**`()`

Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also the corresponding attribute of the derived class of interest.

method

`generic.`**`put`**`()`

Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also the corresponding attribute of the derived class of interest.

method

`generic.`**`ravel`**`()`

Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also the corresponding attribute of the derived class of interest.

method

`generic.`**`repeat`**`()`

Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also the corresponding attribute of the derived class of interest.

method

`generic.`**`reshape`**`()`

Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also the corresponding attribute of the derived class of interest.

method

`generic.`**`resize`**`()`

Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also the corresponding attribute of the derived class of interest.

method

`generic.`**`round`**`()`

Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also the corresponding attribute of the derived class of interest.

method

`generic.`**`searchsorted`**`()`

Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also the corresponding attribute of the derived class of interest.

method

`generic.`**`setfield`**`()`

Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also the corresponding attribute of the derived class of interest.

method

`generic.`**`setflags`**`()`

Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also the corresponding attribute of the derived class of interest.

method

`generic.`**`sort`**`()`

Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also the corresponding attribute of the derived class of interest.

method

`generic.`**`squeeze`**`()`

Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also the corresponding attribute of the derived class of interest.

method

`generic.`**`std`**`()`

Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also the corresponding attribute of the derived class of interest.

method

`generic.`**`sum`**`()`

Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also the corresponding attribute of the derived class of interest.

method

`generic.`**`swapaxes`**`()`

Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also the corresponding attribute of the derived class of interest.

method

`generic.`**`take`**`()`

Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also the corresponding attribute of the derived class of interest.

method

`generic.`**`tofile`**`()`

Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also the corresponding attribute of the derived class of interest.

method

`generic.`**`tolist`**`()`

Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also the corresponding attribute of the derived class of interest.

method

`generic.`**`tostring`**`()`

Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also the corresponding attribute of the derived class of interest.

method

`generic.`**`trace`**`()`

Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also the corresponding attribute of the derived class of interest.

method

`generic.`**`transpose`**`()`

Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also the corresponding attribute of the derived class of interest.

method

`generic.`**`var`**`()`

Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also the corresponding attribute of the derived class of interest.

method

`generic.`**`view`**`()`

Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also the corresponding attribute of the derived class of interest.

| | |
|---|---|
| **conj** | |
| **tobytes** | |

method

`generic.`**`__array__`**`()`

sc.__array__(dtype) return 0-dim array from scalar with specified dtype

method

`generic.`**`__reduce__`**`()`

Helper for pickle.

method

`generic.__setstate__()`

### 1.2.5 Defining new types

There are two ways to effectively define a new array scalar type (apart from composing structured types *dtypes* from the built-in scalar types): One way is to simply subclass the `ndarray` and overwrite the methods of interest. This will work to a degree, but internally certain behaviors are fixed by the data type of the array. To fully customize the data type of an array you need to define a new data-type, and register it with NumPy. Such new types can only be defined in C, using the *NumPy C-API*.

## 1.3 Data type objects (`dtype`)

A data type object (an instance of `numpy.dtype` class) describes how the bytes in the fixed-size block of memory corresponding to an array item should be interpreted. It describes the following aspects of the data:

1. Type of the data (integer, float, Python object, etc.)
2. Size of the data (how many bytes is in *e.g.* the integer)
3. Byte order of the data (little-endian or big-endian)
4. If the data type is structured data type, an aggregate of other data types, (*e.g.*, describing an array item consisting of an integer and a float),
   1. what are the names of the "fields" of the structure, by which they can be *accessed*,
   2. what is the data-type of each field, and
   3. which part of the memory block each field takes.
5. If the data type is a sub-array, what is its shape and data type.

To describe the type of scalar data, there are several *built-in scalar types* in NumPy for various precision of integers, floating-point numbers, *etc*. An item extracted from an array, *e.g.*, by indexing, will be a Python object whose type is the scalar type associated with the data type of the array.

Note that the scalar types are not `dtype` objects, even though they can be used in place of one whenever a data type specification is needed in NumPy.

Structured data types are formed by creating a data type whose field contain other data types. Each field has a name by which it can be *accessed*. The parent data type should be of sufficient size to contain all its fields; the parent is nearly always based on the `void` type which allows an arbitrary item size. Structured data types may also contain nested structured sub-array data types in their fields.

Finally, a data type can describe items that are themselves arrays of items of another data type. These sub-arrays must, however, be of a fixed size.

If an array is created using a data-type describing a sub-array, the dimensions of the sub-array are appended to the shape of the array when the array is created. Sub-arrays in a field of a structured type behave differently, see *Field Access*.

Sub-arrays always have a C-contiguous memory layout.

**Example**

A simple data type containing a 32-bit big-endian integer: (see *Specifying and constructing data types* for details on construction)

```
>>> dt = np.dtype('>i4')
>>> dt.byteorder
'>'
>>> dt.itemsize
4
>>> dt.name
'int32'
>>> dt.type is np.int32
True
```

The corresponding array scalar type is `int32`.

---

**Example**

A structured data type containing a 16-character string (in field 'name') and a sub-array of two 64-bit floating-point number (in field 'grades'):

```
>>> dt = np.dtype([('name', np.unicode_, 16), ('grades', np.float64, (2,))])
>>> dt['name']
dtype('<U16')
>>> dt['grades']
dtype(('<f8', (2,)))
```

Items of an array of this data type are wrapped in an *array scalar* type that also has two fields:

```
>>> x = np.array([('Sarah', (8.0, 7.0)), ('John', (6.0, 7.0))], dtype=dt)
>>> x[1]
('John', [6., 7.])
>>> x[1]['grades']
array([6.,  7.])
>>> type(x[1])
<class 'numpy.void'>
>>> type(x[1]['grades'])
<class 'numpy.ndarray'>
```

## 1.3.1 Specifying and constructing data types

Whenever a data-type is required in a NumPy function or method, either a `dtype` object or something that can be converted to one can be supplied. Such conversions are done by the `dtype` constructor:

| | |
|---|---|
| `dtype`(obj[, align, copy]) | Create a data type object. |

**class** `numpy.`**`dtype`**(*obj*, *align=False*, *copy=False*)

Create a data type object.

A numpy array is homogeneous, and contains elements described by a dtype object. A dtype object can be constructed from different combinations of fundamental numeric types.

**Parameters**

**obj**

Object to be converted to a data type object.

**align**

[bool, optional] Add padding to the fields to match what a C compiler would output for a similar C-struct. Can be `True` only if *obj* is a dictionary or a comma-separated string. If a struct dtype is being created, this also sets a sticky alignment flag `isalignedstruct`.

**copy**

[bool, optional] Make a new copy of the data-type object. If `False`, the result may just be a reference to a built-in data-type object.

**See also:**

*`result_type`*

## Examples

Using array-scalar type:

```
>>> np.dtype(np.int16)
dtype('int16')
```

Structured type, one field name 'f1', containing int16:

```
>>> np.dtype([('f1', np.int16)])
dtype([('f1', '<i2')])
```

Structured type, one field named 'f1', in itself containing a structured type with one field:

```
>>> np.dtype([('f1', [('f1', np.int16)])])
dtype([('f1', [('f1', '<i2')])])
```

Structured type, two fields: the first field contains an unsigned int, the second an int32:

```
>>> np.dtype([('f1', np.uint64), ('f2', np.int32)])
dtype([('f1', '<u8'), ('f2', '<i4')])
```

Using array-protocol type strings:

```
>>> np.dtype([('a','f8'),('b','S10')])
dtype([('a', '<f8'), ('b', 'S10')])
```

Using comma-separated field formats. The shape is (2,3):

```
>>> np.dtype("i4, (2,3)f8")
dtype([('f0', '<i4'), ('f1', '<f8', (2, 3))])
```

Using tuples. `int` is a fixed type, 3 the field's shape. `void` is a flexible type, here of size 10:

```
>>> np.dtype([('hello',(np.int64,3)),('world',np.void,10)])
dtype([('hello', '<i8', (3,)), ('world', 'V10')])
```

Subdivide `int16` into 2 `int8`'s, called x and y. 0 and 1 are the offsets in bytes:

```
>>> np.dtype((np.int16, {'x':(np.int8,0), 'y':(np.int8,1)}))
dtype((numpy.int16, [('x', 'i1'), ('y', 'i1')]))
```

Using dictionaries. Two fields named 'gender' and 'age':

```
>>> np.dtype({'names':['gender','age'], 'formats':['S1',np.uint8]})
dtype([('gender', 'S1'), ('age', 'u1')])
```

Offsets in bytes, here 0 and 25:

```
>>> np.dtype({'surname':('S25',0),'age':(np.uint8,25)})
dtype([('surname', 'S25'), ('age', 'u1')])
```

**Attributes**

***`alignment`***

The required alignment (bytes) of this data-type according to the compiler.

***`base`***

Returns dtype for the base element of the subarrays, regardless of their dimension or shape.

***`byteorder`***

A character indicating the byte-order of this data-type object.

***`char`***

A unique character code for each of the 21 different built-in types.

***`descr`***

*__array_interface__* description of the data-type.

***`fields`***

Dictionary of named fields defined for this data type, or `None`.

***`flags`***

Bit-flags describing how this data type is to be interpreted.

***`hasobject`***

Boolean indicating whether this dtype contains any reference-counted objects in any fields or sub-dtypes.

**`isalignedstruct`**

Boolean indicating whether the dtype is a struct which maintains field alignment.

***`isbuiltin`***

Integer indicating how this dtype relates to the built-in dtypes.

***`isnative`***

Boolean indicating whether the byte order of this dtype is native to the platform.

***`itemsize`***

The element size of this data-type object.

***`kind`***

A character code (one of 'biufcmMOSUV') identifying the general kind of data.

**metadata**

***name***

A bit-width name for this data-type.

***names***

Ordered list of field names, or `None` if there are no fields.

**ndim**

Number of dimensions of the sub-array if this data type describes a sub-array, and `0` otherwise.

***num***

A unique number for each of the 21 different built-in types.

***shape***

Shape tuple of the sub-array if this data type describes a sub-array, and `()` otherwise.

***str***

The array-protocol typestring of this data-type object.

***subdtype***

Tuple `(item_dtype, shape)` if this *dtype* describes a sub-array, and None otherwise.

***type***

The type object used to instantiate a scalar of this data-type.

### Methods

| | |
|---|---|
| *newbyteorder*([new_order]) | Return a new dtype with a different byte order. |

method

`dtype.`**`newbyteorder`**(*new_order='S'*)

Return a new dtype with a different byte order.

Changes are also made in all fields and sub-arrays of the data type.

**Parameters**

**new_order**

[string, optional] Byte order to force; a value from the byte order specifications below. The default value ('S') results in swapping the current byte order. *new_order* codes can be any of:

- 'S' - swap dtype from current to opposite endian
- {'<', 'L'} - little endian
- {'>', 'B'} - big endian
- {'=', 'N'} - native order
- {'|', 'I'} - ignore (no change to byte order)

The code does a case-insensitive check on the first letter of *new_order* for these alternatives. For example, any of '>' or 'B' or 'b' or 'brian' are valid to specify big-endian.

**Returns**

**new_dtype**

[dtype] New dtype object with the given change to the byte order.

### Notes

Changes are also made in all fields and sub-arrays of the data type.

### Examples

```
>>> import sys
>>> sys_is_le = sys.byteorder == 'little'
>>> native_code = sys_is_le and '<' or '>'
>>> swapped_code = sys_is_le and '>' or '<'
>>> native_dt = np.dtype(native_code+'i2')
>>> swapped_dt = np.dtype(swapped_code+'i2')
>>> native_dt.newbyteorder('S') == swapped_dt
True
>>> native_dt.newbyteorder() == swapped_dt
True
>>> native_dt == swapped_dt.newbyteorder('S')
True
>>> native_dt == swapped_dt.newbyteorder('=')
True
>>> native_dt == swapped_dt.newbyteorder('N')
True
>>> native_dt == native_dt.newbyteorder('|')
True
>>> np.dtype('<i2') == native_dt.newbyteorder('<')
True
>>> np.dtype('<i2') == native_dt.newbyteorder('L')
True
>>> np.dtype('>i2') == native_dt.newbyteorder('>')
True
>>> np.dtype('>i2') == native_dt.newbyteorder('B')
True
```

What can be converted to a data-type object is described below:

*`dtype`* object

Used as-is.

None

The default data type: `float_`.

Array-scalar types

The 24 built-in *array scalar type objects* all convert to an associated data-type object. This is true for their sub-classes as well.

Note that not all data-type information can be supplied with a type-object: for example, `flexible` data-types have a default *itemsize* of 0, and require an explicitly given size to be useful.

---

**Example**

```
>>> dt = np.dtype(np.int32)       # 32-bit integer
>>> dt = np.dtype(np.complex128) # 128-bit complex floating-point number
```

---

Generic types

The generic hierarchical type objects convert to corresponding type objects according to the associations:

| | |
|---|---|
| `number`, `inexact`, `floating` | `float` |
| `complexfloating` | `cfloat` |
| `integer`, `signedinteger` | `int_` |
| `unsignedinteger` | `uint` |
| `character` | `string` |
| *`generic`*, `flexible` | `void` |

Built-in Python types

Several python types are equivalent to a corresponding array scalar when used to generate a *`dtype`* object:

| | |
|---|---|
| `int` | `int_` |
| `bool` | `bool_` |
| `float` | `float_` |
| `complex` | `cfloat` |
| `bytes` | `bytes_` |
| `str` | `str_` |
| `buffer` | `void` |
| (all others) | `object_` |

Note that `str` refers to either null terminated bytes or unicode strings depending on the Python version. In code targeting both Python 2 and 3 `np.unicode_` should be used as a dtype for strings. See *Note on string types*.

---

**Example**

```
>>> dt = np.dtype(float)   # Python-compatible floating-point number
>>> dt = np.dtype(int)     # Python-compatible integer
>>> dt = np.dtype(object)  # Python object
```

---

**Note:** All other types map to `object_` for convenience. Code should expect that such types may map to a specific (new) dtype in future the future.

Types with `.dtype`

Any type object with a `dtype` attribute: The attribute will be accessed and used directly. The attribute must return something that is convertible into a dtype object.

Several kinds of strings can be converted. Recognized strings can be prepended with `'>'` (big-endian), `'<'` (little-endian), or `'='` (hardware-native, the default), to specify the byte order.

One-character strings

Each built-in data-type has a character code (the updated Numeric typecodes), that uniquely identifies it.

**Example**

```
>>> dt = np.dtype('b')  # byte, native byte order
>>> dt = np.dtype('>H') # big-endian unsigned short
>>> dt = np.dtype('<f') # little-endian single-precision float
>>> dt = np.dtype('d')  # double-precision floating-point number
```

Array-protocol type strings (see *The Array Interface*)

The first character specifies the kind of data and the remaining characters specify the number of bytes per item, except for Unicode, where it is interpreted as the number of characters. The item size must correspond to an existing type, or an error will be raised. The supported kinds are

| | |
|---|---|
| `'?'` | boolean |
| `'b'` | (signed) byte |
| `'B'` | unsigned byte |
| `'i'` | (signed) integer |
| `'u'` | unsigned integer |
| `'f'` | floating-point |
| `'c'` | complex-floating point |
| `'m'` | timedelta |
| `'M'` | datetime |
| `'O'` | (Python) objects |
| `'S'`, `'a'` | zero-terminated bytes (not recommended) |
| `'U'` | Unicode string |
| `'V'` | raw data (`void`) |

**Example**

```
>>> dt = np.dtype('i4')   # 32-bit signed integer
>>> dt = np.dtype('f8')   # 64-bit floating-point number
>>> dt = np.dtype('c16')  # 128-bit complex floating-point number
>>> dt = np.dtype('a25')  # 25-length zero-terminated bytes
>>> dt = np.dtype('U25')  # 25-character string
```

**Note on string types**

For backward compatibility with Python 2 the `S` and `a` typestrings remain zero-terminated bytes and `np.string_` continues to map to `np.bytes_`. To use actual strings in Python 3 use `U` or `np.unicode_`. For signed bytes that do not need zero-termination `b` or `i1` can be used.

String with comma-separated fields

A short-hand notation for specifying the format of a structured data type is a comma-separated string of basic formats.

A basic format in this context is an optional shape specifier followed by an array-protocol type string. Parenthesis are required on the shape if it has more than one dimension. NumPy allows a modification on the format in that any string that can uniquely identify the type can be used to specify the data-type in a field. The generated data-type fields are named `'f0'`, `'f1'`, …, `'f<N-1>'` where N (>1) is the number of comma-separated basic formats in the string. If the optional shape specifier is provided, then the data-type for the corresponding field describes a sub-array.

---

**Example**

- field named `f0` containing a 32-bit integer
- field named `f1` containing a 2 x 3 sub-array of 64-bit floating-point numbers
- field named `f2` containing a 32-bit floating-point number

```
>>> dt = np.dtype("i4, (2,3)f8, f4")
```

- field named `f0` containing a 3-character string
- field named `f1` containing a sub-array of shape (3,) containing 64-bit unsigned integers
- field named `f2` containing a 3 x 4 sub-array containing 10-character strings

```
>>> dt = np.dtype("a3, 3u8, (3,4)a10")
```

---

Type strings

Any string in `numpy.sctypeDict`.keys():

---

**Example**

```
>>> dt = np.dtype('uint32')    # 32-bit unsigned integer
>>> dt = np.dtype('float64')   # 64-bit floating-point number
```

---

`(flexible_dtype, itemsize)`

The first argument must be an object that is converted to a zero-sized flexible data-type object, the second argument is an integer providing the desired itemsize.

---

**Example**

```
>>> dt = np.dtype((np.void, 10))  # 10-byte wide data block
>>> dt = np.dtype(('U', 10))   # 10-character unicode string
```

---

`(fixed_dtype, shape)`

The first argument is any object that can be converted into a fixed-size data-type object. The second argument is the desired shape of this type. If the shape parameter is 1, then the data-type object used to be equivalent to fixed dtype. This behaviour is deprecated since NumPy 1.17 and will raise an error in the future. If *shape* is a tuple, then the new dtype defines a sub-array of the given shape.

---

**Example**

```
>>> dt = np.dtype((np.int32, (2,2)))          # 2 x 2 integer sub-array
>>> dt = np.dtype(('i4, (2,3)f8, f4', (2,3))) # 2 x 3 structured sub-array
```

---

`[(field_name, field_dtype, field_shape), ...]`

*obj* should be a list of fields where each field is described by a tuple of length 2 or 3. (Equivalent to the `descr` item in the `__array_interface__` attribute.)

The first element, *field_name*, is the field name (if this is `''` then a standard field name, `'f#'`, is assigned). The field name may also be a 2-tuple of strings where the first string is either a "title" (which may be any string or unicode string) or meta-data for the field which can be any object, and the second string is the "name" which must be a valid Python identifier.

The second element, *field_dtype*, can be anything that can be interpreted as a data-type.

The optional third element *field_shape* contains the shape if this field represents an array of the data-type in the second element. Note that a 3-tuple with a third argument equal to 1 is equivalent to a 2-tuple.

This style does not accept *align* in the `dtype` constructor as it is assumed that all of the memory is accounted for by the array interface description.

---

**Example**

Data-type with fields `big` (big-endian 32-bit integer) and `little` (little-endian 32-bit integer):

```
>>> dt = np.dtype([('big', '>i4'), ('little', '<i4')])
```

Data-type with fields `R`, `G`, `B`, `A`, each being an unsigned 8-bit integer:

```
>>> dt = np.dtype([('R','u1'), ('G','u1'), ('B','u1'), ('A','u1')])
```

---

`{'names': ..., 'formats': ..., 'offsets': ..., 'titles': ..., 'itemsize': ...}`

This style has two required and three optional keys. The *names* and *formats* keys are required. Their respective values are equal-length lists with the field names and the field formats. The field names must be strings and the field formats can be any object accepted by `dtype` constructor.

When the optional keys *offsets* and *titles* are provided, their values must each be lists of the same length as the *names* and *formats* lists. The *offsets* value is a list of byte offsets (limited to `ctypes.c_int`) for each field, while the *titles* value is a list of titles for each field (None can be used if no title is desired for that field). The *titles* can be any `string` or `unicode` object and will add another entry to the fields dictionary keyed by the title and referencing the same field tuple which will contain the title as an additional tuple member.

The *itemsize* key allows the total size of the dtype to be set, and must be an integer large enough so all the fields are within the dtype. If the dtype being constructed is aligned, the *itemsize* must also be divisible by the struct alignment. Total dtype *itemsize* is limited to `ctypes.c_int`.

---

**Example**

Data type with fields `r`, `g`, `b`, `a`, each being an 8-bit unsigned integer:

```
>>> dt = np.dtype({'names': ['r','g','b','a'],
...                'formats': [np.uint8, np.uint8, np.uint8, np.uint8]})
```

Data type with fields `r` and `b` (with the given titles), both being 8-bit unsigned integers, the first at byte position 0 from the start of the field and the second at position 2:

```
>>> dt = np.dtype({'names': ['r','b'], 'formats': ['u1', 'u1'],
...                'offsets': [0, 2],
...                'titles': ['Red pixel', 'Blue pixel']})
```

---

`{'field1': ..., 'field2': ..., ...}`

This usage is discouraged, because it is ambiguous with the other dict-based construction method. If you have a field called 'names' and a field called 'formats' there will be a conflict.

This style allows passing in the *`fields`* attribute of a data-type object.

*obj* should contain string or unicode keys that refer to `(data-type, offset)` or `(data-type, offset, title)` tuples.

---

**Example**

Data type containing field `col1` (10-character string at byte position 0), `col2` (32-bit float at byte position 10), and `col3` (integers at byte position 14):

```
>>> dt = np.dtype({'col1': ('U10', 0), 'col2': (np.float32, 10),
...                'col3': (int, 14)})
```

---

`(base_dtype, new_dtype)`

In NumPy 1.7 and later, this form allows *base_dtype* to be interpreted as a structured dtype. Arrays created with this dtype will have underlying dtype *base_dtype* but will have fields and flags taken from *new_dtype*. This is useful for creating custom structured dtypes, as done in *record arrays*.

This form also makes it possible to specify struct dtypes with overlapping fields, functioning like the 'union' type in C. This usage is discouraged, however, and the union mechanism is preferred.

Both arguments must be convertible to data-type objects with the same total size.

---

**Example**

32-bit integer, whose first two bytes are interpreted as an integer via field `real`, and the following two bytes via field `imag`.

```
>>> dt = np.dtype((np.int32,{'real':(np.int16, 0),'imag':(np.int16, 2)}))
```

32-bit integer, which is interpreted as consisting of a sub-array of shape `(4,)` containing 8-bit integers:

```
>>> dt = np.dtype((np.int32, (np.int8, 4)))
```

32-bit integer, containing fields `r`, `g`, `b`, `a` that interpret the 4 bytes in the integer as four unsigned integers:

```
>>> dt = np.dtype(('i4', [('r','u1'),('g','u1'),('b','u1'),('a','u1')]))
```

---

## 1.3.2 `dtype`

NumPy data type descriptions are instances of the *`dtype`* class.

### Attributes

The type of the data is described by the following *dtype* attributes:

| | |
|---|---|
| *dtype.type* | The type object used to instantiate a scalar of this data-type. |
| *dtype.kind* | A character code (one of 'biufcmMOSUV') identifying the general kind of data. |
| *dtype.char* | A unique character code for each of the 21 different built-in types. |
| *dtype.num* | A unique number for each of the 21 different built-in types. |
| *dtype.str* | The array-protocol typestring of this data-type object. |

attribute

`dtype.`**`type`**
The type object used to instantiate a scalar of this data-type.

attribute

`dtype.`**`kind`**
A character code (one of 'biufcmMOSUV') identifying the general kind of data.

| | |
|---|---|
| b | boolean |
| i | signed integer |
| u | unsigned integer |
| f | floating-point |
| c | complex floating-point |
| m | timedelta |
| M | datetime |
| O | object |
| S | (byte-)string |
| U | Unicode |
| V | void |

### Examples

```
>>> dt = np.dtype('i4')
>>> dt.kind
'i'
>>> dt = np.dtype('f8')
>>> dt.kind
'f'
>>> dt = np.dtype([('field1', 'f8')])
>>> dt.kind
'V'
```

attribute

`dtype.`**`char`**
A unique character code for each of the 21 different built-in types.

### Examples

```
>>> x = np.dtype(float)
>>> x.char
'd'
```

attribute

`dtype.`**`num`**

A unique number for each of the 21 different built-in types.

These are roughly ordered from least-to-most precision.

### Examples

```
>>> dt = np.dtype(str)
>>> dt.num
19
```

```
>>> dt = np.dtype(float)
>>> dt.num
12
```

attribute

`dtype.`**`str`**

The array-protocol typestring of this data-type object.

Size of the data is in turn described by:

| | |
|---|---|
| *`dtype.name`* | A bit-width name for this data-type. |
| *`dtype.itemsize`* | The element size of this data-type object. |

attribute

`dtype.`**`name`**

A bit-width name for this data-type.

Un-sized flexible data-type objects do not have this attribute.

### Examples

```
>>> x = np.dtype(float)
>>> x.name
'float64'
>>> x = np.dtype([('a', np.int32, 8), ('b', np.float64, 6)])
>>> x.name
'void640'
```

attribute

`dtype.`**`itemsize`**

The element size of this data-type object.

For 18 of the 21 types this number is fixed by the data-type. For the flexible data-types, this number can be anything.

### Examples

```
>>> arr = np.array([[1, 2], [3, 4]])
>>> arr.dtype
dtype('int64')
>>> arr.itemsize
8
```

```
>>> dt = np.dtype([('name', np.str_, 16), ('grades', np.float64, (2,))])
>>> dt.itemsize
80
```

Endianness of this data:

| | |
|---|---|
| *dtype.byteorder* | A character indicating the byte-order of this data-type object. |

attribute

`dtype.`**`byteorder`**

A character indicating the byte-order of this data-type object.

One of:

| | |
|---|---|
| '=' | native |
| '<' | little-endian |
| '>' | big-endian |
| '\|' | not applicable |

All built-in data-type objects have byteorder either '=' or '|'.

### Examples

```
>>> dt = np.dtype('i2')
>>> dt.byteorder
'='
>>> # endian is not relevant for 8 bit numbers
>>> np.dtype('i1').byteorder
'|'
>>> # or ASCII strings
>>> np.dtype('S2').byteorder
'|'
>>> # Even if specific code is given, and it is native
>>> # '=' is the byteorder
>>> import sys
>>> sys_is_le = sys.byteorder == 'little'
>>> native_code = sys_is_le and '<' or '>'
>>> swapped_code = sys_is_le and '>' or '<'
>>> dt = np.dtype(native_code + 'i2')
>>> dt.byteorder
'='
>>> # Swapped code shows up as itself
>>> dt = np.dtype(swapped_code + 'i2')
```

(continues on next page)

(continued from previous page)

```
>>> dt.byteorder == swapped_code
True
```

Information about sub-data-types in a structured data type:

| | |
|---|---|
| `dtype.fields` | Dictionary of named fields defined for this data type, or `None`. |
| `dtype.names` | Ordered list of field names, or `None` if there are no fields. |

attribute

`dtype.`**`fields`**

Dictionary of named fields defined for this data type, or `None`.

The dictionary is indexed by keys that are the names of the fields. Each entry in the dictionary is a tuple fully describing the field:

```
(dtype, offset[, title])
```

Offset is limited to C int, which is signed and usually 32 bits. If present, the optional title can be any object (if it is a string or unicode then it will also be a key in the fields dictionary, otherwise it's meta-data). Notice also that the first two elements of the tuple can be passed directly as arguments to the `ndarray.getfield` and `ndarray.setfield` methods.

**See also:**

*`ndarray.getfield`*, *`ndarray.setfield`*

### Examples

```
>>> dt = np.dtype([('name', np.str_, 16), ('grades', np.float64, (2,))])
>>> print(dt.fields)
{'grades': (dtype(('float64',(2,))), 16), 'name': (dtype('|S16'), 0)}
```

attribute

`dtype.`**`names`**

Ordered list of field names, or `None` if there are no fields.

The names are ordered according to increasing byte offset. This can be used, for example, to walk through all of the named fields in offset order.

### Examples

```
>>> dt = np.dtype([('name', np.str_, 16), ('grades', np.float64, (2,))])
>>> dt.names
('name', 'grades')
```

For data types that describe sub-arrays:

| | |
|---|---|
| `dtype.subdtype` | Tuple `(item_dtype, shape)` if this `dtype` describes a sub-array, and None otherwise. |

Continued on next page

Table 31 – continued from previous page

| | |
|---|---|
| *dtype.shape* | Shape tuple of the sub-array if this data type describes a sub-array, and `()` otherwise. |

attribute

`dtype.`**`subdtype`**

Tuple `(item_dtype, shape)` if this *dtype* describes a sub-array, and None otherwise.

The *shape* is the fixed shape of the sub-array described by this data type, and *item_dtype* the data type of the array.

If a field whose dtype object has this attribute is retrieved, then the extra dimensions implied by *shape* are tacked on to the end of the retrieved array.

**See also:**

*dtype.base*

### Examples

```
>>> x = numpy.dtype('8f')
>>> x.subdtype
(dtype('float32'), (8,))
```

```
>>> x =  numpy.dtype('i2')
>>> x.subdtype
>>>
```

attribute

`dtype.`**`shape`**

Shape tuple of the sub-array if this data type describes a sub-array, and `()` otherwise.

### Examples

```
>>> dt = np.dtype(('i4', 4))
>>> dt.shape
(4,)
```

```
>>> dt = np.dtype(('i4', (2, 3)))
>>> dt.shape
(2, 3)
```

Attributes providing additional information:

| | |
|---|---|
| *dtype.hasobject* | Boolean indicating whether this dtype contains any reference-counted objects in any fields or sub-dtypes. |
| *dtype.flags* | Bit-flags describing how this data type is to be interpreted. |
| *dtype.isbuiltin* | Integer indicating how this dtype relates to the built-in dtypes. |
| *dtype.isnative* | Boolean indicating whether the byte order of this dtype is native to the platform. |
| *dtype.descr* | *__array_interface__* description of the data-type. |

Continued on next page

Table 32 – continued from previous page

| | |
|---|---|
| `dtype.alignment` | The required alignment (bytes) of this data-type according to the compiler. |
| `dtype.base` | Returns dtype for the base element of the subarrays, regardless of their dimension or shape. |

attribute

`dtype.`**`hasobject`**

Boolean indicating whether this dtype contains any reference-counted objects in any fields or sub-dtypes.

Recall that what is actually in the ndarray memory representing the Python object is the memory address of that object (a pointer). Special handling may be required, and this attribute is useful for distinguishing data types that may contain arbitrary Python objects and data-types that won't.

attribute

`dtype.`**`flags`**

Bit-flags describing how this data type is to be interpreted.

Bit-masks are in `numpy.core.multiarray` as the constants *ITEM_HASOBJECT*, *LIST_PICKLE*, *ITEM_IS_POINTER*, *NEEDS_INIT*, *NEEDS_PYAPI*, *USE_GETITEM*, *USE_SETITEM*. A full explanation of these flags is in C-API documentation; they are largely useful for user-defined data-types.

The following example demonstrates that operations on this particular dtype requires Python C-API.

## Examples

```
>>> x = np.dtype([('a', np.int32, 8), ('b', np.float64, 6)])
>>> x.flags
16
>>> np.core.multiarray.NEEDS_PYAPI
16
```

attribute

`dtype.`**`isbuiltin`**

Integer indicating how this dtype relates to the built-in dtypes.

Read-only.

| | |
|---|---|
| 0 | if this is a structured array type, with fields |
| 1 | if this is a dtype compiled into numpy (such as ints, floats etc) |
| 2 | if the dtype is for a user-defined numpy type A user-defined type uses the numpy C-API machinery to extend numpy to handle a new array type. See user.user-defined-data-types in the NumPy manual. |

## Examples

```
>>> dt = np.dtype('i2')
>>> dt.isbuiltin
1
>>> dt = np.dtype('f8')
>>> dt.isbuiltin
1
>>> dt = np.dtype([('field1', 'f8')])
```

(continues on next page)

(continued from previous page)

```
>>> dt.isbuiltin
0
```

attribute

**`dtype.isnative`**

Boolean indicating whether the byte order of this dtype is native to the platform.

attribute

**`dtype.descr`**

*__array_interface__* description of the data-type.

The format is that required by the 'descr' key in the *__array_interface__* attribute.

Warning: This attribute exists specifically for *__array_interface__*, and passing it directly to *np.dtype* will not accurately reconstruct some dtypes (e.g., scalar and subarray dtypes).

### Examples

```
>>> x = np.dtype(float)
>>> x.descr
[('', '<f8')]
```

```
>>> dt = np.dtype([('name', np.str_, 16), ('grades', np.float64, (2,))])
>>> dt.descr
[('name', '<U16'), ('grades', '<f8', (2,))]
```

attribute

**`dtype.alignment`**

The required alignment (bytes) of this data-type according to the compiler.

More information is available in the C-API section of the manual.

### Examples

```
>>> x = np.dtype('i4')
>>> x.alignment
4
```

```
>>> x = np.dtype(float)
>>> x.alignment
8
```

attribute

**`dtype.base`**

Returns dtype for the base element of the subarrays, regardless of their dimension or shape.

**See also:**

*`dtype.subdtype`*

### Examples

```
>>> x = numpy.dtype('8f')
>>> x.base
dtype('float32')
```

```
>>> x =  numpy.dtype('i2')
>>> x.base
dtype('int16')
```

### Methods

Data types have the following method for changing the byte order:

| | |
|---|---|
| *`dtype.newbyteorder`*([new_order]) | Return a new dtype with a different byte order. |

The following methods implement the pickle protocol:

| | |
|---|---|
| *`dtype.__reduce__`*() | Helper for pickle. |
| *`dtype.__setstate__`*() | |

method

`dtype.`**`__reduce__`**`()`
Helper for pickle.

method

`dtype.`**`__setstate__`**`()`

## 1.4 Indexing

**See also:**

Indexing basics

*`ndarrays`* can be indexed using the standard Python `x[obj]` syntax, where *x* is the array and *obj* the selection. There are three kinds of indexing available: field access, basic slicing, advanced indexing. Which one occurs depends on *obj*.

**Note:** In Python, `x[(exp1, exp2, ..., expN)]` is equivalent to `x[exp1, exp2, ..., expN]`; the latter is just syntactic sugar for the former.

### 1.4.1 Basic Slicing and Indexing

Basic slicing extends Python's basic concept of slicing to N dimensions. Basic slicing occurs when *obj* is a `slice` object (constructed by `start:stop:step` notation inside of brackets), an integer, or a tuple of slice objects and integers. `Ellipsis` and `newaxis` objects can be interspersed with these as well.

Deprecated since version 1.15.0: In order to remain backward compatible with a common usage in Numeric, basic slicing is also initiated if the selection object is any non-ndarray and non-tuple sequence (such as a `list`) containing `slice` objects, the `Ellipsis` object, or the `newaxis` object, but not for integer arrays or other embedded sequences.

The simplest case of indexing with *N* integers returns an *array scalar* representing the corresponding item. As in Python, all indices are zero-based: for the *i*-th index $n_i$, the valid range is $0 \le n_i < d_i$ where $d_i$ is the *i*-th element of the shape of the array. Negative indices are interpreted as counting from the end of the array (*i.e.*, if $n_i < 0$, it means $n_i + d_i$).

All arrays generated by basic slicing are always views of the original array.

**Note:** NumPy slicing creates a view instead of a copy as in the case of builtin Python sequences such as string, tuple and list. Care must be taken when extracting a small portion from a large array which becomes useless after the extraction, because the small portion extracted contains a reference to the large original array whose memory will not be released until all arrays derived from it are garbage-collected. In such cases an explicit `copy()` is recommended.

The standard rules of sequence slicing apply to basic slicing on a per-dimension basis (including using a step index). Some useful concepts to remember include:

- The basic slice syntax is `i:j:k` where *i* is the starting index, *j* is the stopping index, and *k* is the step ($k \neq 0$). This selects the *m* elements (in the corresponding dimension) with index values *i*, *i + k*, …, *i + (m - 1) k* where $m = q + (r \neq 0)$ and *q* and *r* are the quotient and remainder obtained by dividing *j - i* by *k*: *j - i = q k + r*, so that *i + (m - 1) k < j*.

**Example**

```
>>> x = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> x[1:7:2]
array([1, 3, 5])
```

- Negative *i* and *j* are interpreted as *n + i* and *n + j* where *n* is the number of elements in the corresponding dimension. Negative *k* makes stepping go towards smaller indices.

**Example**

```
>>> x[-2:10]
array([8, 9])
>>> x[-3:3:-1]
array([7, 6, 5, 4])
```

- Assume *n* is the number of elements in the dimension being sliced. Then, if *i* is not given it defaults to 0 for *k > 0* and *n - 1* for *k < 0* . If *j* is not given it defaults to *n* for *k > 0* and *-n-1* for *k < 0* . If *k* is not given it defaults to 1. Note that `::` is the same as `:` and means select all indices along this axis.

**Example**

```
>>> x[5:]
array([5, 6, 7, 8, 9])
```

- If the number of objects in the selection tuple is less than *N*, then `:` is assumed for any subsequent dimensions.

**Example**

```
>>> x = np.array([[[1],[2],[3]], [[4],[5],[6]]])
>>> x.shape
(2, 3, 1)
>>> x[1:2]
array([[[4],
        [5],
        [6]]])
```

- `Ellipsis` expands to the number of `:` objects needed for the selection tuple to index all dimensions. In most cases, this means that length of the expanded selection tuple is `x.ndim`. There may only be a single ellipsis present.

**Example**

```
>>> x[...,0]
array([[1, 2, 3],
       [4, 5, 6]])
```

- Each *`newaxis`* object in the selection tuple serves to expand the dimensions of the resulting selection by one unit-length dimension. The added dimension is the position of the *`newaxis`* object in the selection tuple.

**Example**

```
>>> x[:,np.newaxis,:,:].shape
(2, 1, 3, 1)
```

- An integer, *i*, returns the same values as `i:i+1` **except** the dimensionality of the returned object is reduced by 1. In particular, a selection tuple with the *p*-th element an integer (and all other entries `:`) returns the corresponding sub-array with dimension *N - 1*. If *N = 1* then the returned object is an array scalar. These objects are explained in *Scalars*.
- If the selection tuple has all entries `:` except the *p*-th entry which is a slice object `i:j:k`, then the returned array has dimension *N* formed by concatenating the sub-arrays returned by integer indexing of elements *i*, *i+k*, …, *i + (m - 1) k < j*,
- Basic slicing with more than one non-`:` entry in the slicing tuple, acts like repeated application of slicing using a single non-`:` entry, where the non-`:` entries are successively taken (with all other non-`:` entries replaced by `:`). Thus, `x[ind1,...,ind2,:]` acts like `x[ind1][...,ind2,:]` under basic slicing.

> **Warning:** The above is **not** true for advanced indexing.

- You may use slicing to set values in the array, but (unlike lists) you can never grow the array. The size of the value to be set in `x[obj] = value` must be (broadcastable) to the same shape as `x[obj]`.

---

**Note:** Remember that a slicing tuple can always be constructed as *obj* and used in the `x[obj]` notation. Slice objects can be used in the construction in place of the `[start:stop:step]` notation. For example, `x[1:10:5,::-1]` can also be implemented as `obj = (slice(1,10,5), slice(None,None,-1)); x[obj]` . This can be useful for constructing generic code that works on arrays of arbitrary dimension.

---

`numpy.`**`newaxis`**

The *`newaxis`* object can be used in all slicing operations to create an axis of length one. *`newaxis`* is an alias for 'None', and 'None' can be used in place of this with the same result.

## 1.4.2 Advanced Indexing

Advanced indexing is triggered when the selection object, *obj*, is a non-tuple sequence object, an *`ndarray`* (of data type integer or bool), or a tuple with at least one sequence object or ndarray (of data type integer or bool). There are two types of advanced indexing: integer and Boolean.

Advanced indexing always returns a *copy* of the data (contrast with basic slicing that returns a view).

**Warning:** The definition of advanced indexing means that `x[(1,2,3),]` is fundamentally different than `x[(1,2,3)]`. The latter is equivalent to `x[1,2,3]` which will trigger basic selection while the former will trigger advanced indexing. Be sure to understand why this occurs.

Also recognize that `x[[1,2,3]]` will trigger advanced indexing, whereas due to the deprecated Numeric compatibility mentioned above, `x[[1,2,slice(None)]]` will trigger basic slicing.

### Integer array indexing

Integer array indexing allows selection of arbitrary items in the array based on their *N*-dimensional index. Each integer array represents a number of indexes into that dimension.

### Purely integer array indexing

When the index consists of as many integer arrays as the array being indexed has dimensions, the indexing is straight forward, but different from slicing.

Advanced indexes always are *broadcast* and iterated as *one*:

```
result[i_1, ..., i_M] == x[ind_1[i_1, ..., i_M], ind_2[i_1, ..., i_M],
                           ..., ind_N[i_1, ..., i_M]]
```

Note that the result shape is identical to the (broadcast) indexing array shapes `ind_1, ..., ind_N`.

---

**Example**

From each row, a specific element should be selected. The row index is just `[0, 1, 2]` and the column index specifies the element to choose for the corresponding row, here `[0, 1, 0]`. Using both together the task can be solved using advanced indexing:

```
>>> x = np.array([[1, 2], [3, 4], [5, 6]])
>>> x[[0, 1, 2], [0, 1, 0]]
array([1, 4, 5])
```

To achieve a behaviour similar to the basic slicing above, broadcasting can be used. The function *`ix_`* can help with this broadcasting. This is best understood with an example.

---

**Example**

From a 4x3 array the corner elements should be selected using advanced indexing. Thus all elements for which the column is one of `[0, 2]` and the row is one of `[0, 3]` need to be selected. To use advanced indexing one needs to select all elements *explicitly*. Using the method explained previously one could write:

```
>>> x = np.array([[ 0,  1,  2],
...               [ 3,  4,  5],
...               [ 6,  7,  8],
...               [ 9, 10, 11]])
>>> rows = np.array([[0, 0],
...                  [3, 3]], dtype=np.intp)
>>> columns = np.array([[0, 2],
...                     [0, 2]], dtype=np.intp)
>>> x[rows, columns]
array([[ 0,  2],
       [ 9, 11]])
```

However, since the indexing arrays above just repeat themselves, broadcasting can be used (compare operations such as `rows[:, np.newaxis] + columns`) to simplify this:

```
>>> rows = np.array([0, 3], dtype=np.intp)
>>> columns = np.array([0, 2], dtype=np.intp)
>>> rows[:, np.newaxis]
array([[0],
       [3]])
>>> x[rows[:, np.newaxis], columns]
array([[ 0,  2],
       [ 9, 11]])
```

This broadcasting can also be achieved using the function *`ix_`*:

```
>>> x[np.ix_(rows, columns)]
array([[ 0,  2],
       [ 9, 11]])
```

Note that without the `np.ix_` call, only the diagonal elements would be selected, as was used in the previous example. This difference is the most important thing to remember about indexing with multiple advanced indexes.

---

### Combining advanced and basic indexing

When there is at least one slice (`:`), ellipsis (`...`) or *`newaxis`* in the index (or the array has more dimensions than there are advanced indexes), then the behaviour can be more complicated. It is like concatenating the indexing result for each advanced index element

In the simplest case, there is only a *single* advanced index. A single advanced index can for example replace a slice and the result array will be the same, however, it is a copy and may have a different memory layout. A slice is preferable when it is possible.

---

**Example**

```
>>> x[1:2, 1:3]
array([[4, 5]])
>>> x[1:2, [1, 2]]
array([[4, 5]])
```

The easiest way to understand the situation may be to think in terms of the result shape. There are two parts to the indexing operation, the subspace defined by the basic indexing (excluding integers) and the subspace from the advanced indexing part. Two cases of index combination need to be distinguished:

- The advanced indexes are separated by a slice, `Ellipsis` or *`newaxis`*. For example `x[arr1, :, arr2]`.
- The advanced indexes are all next to each other. For example `x[..., arr1, arr2, :]` but *not* `x[arr1, :, 1]` since `1` is an advanced index in this regard.

In the first case, the dimensions resulting from the advanced indexing operation come first in the result array, and the subspace dimensions after that. In the second case, the dimensions from the advanced indexing operations are inserted into the result array at the same spot as they were in the initial array (the latter logic is what makes simple advanced indexing behave just like slicing).

**Example**

Suppose `x.shape` is (10,20,30) and `ind` is a (2,3,4)-shaped indexing `intp` array, then `result = x[...,ind,:]` has shape (10,2,3,4,30) because the (20,)-shaped subspace has been replaced with a (2,3,4)-shaped broadcasted indexing subspace. If we let *i, j, k* loop over the (2,3,4)-shaped subspace then `result[...,i,j,k,:] = x[...,ind[i,j,k],:]`. This example produces the same result as *`x.take(ind, axis=-2)`*.

**Example**

Let `x.shape` be (10,20,30,40,50) and suppose `ind_1` and `ind_2` can be broadcast to the shape (2,3,4). Then `x[:,ind_1,ind_2]` has shape (10,2,3,4,40,50) because the (20,30)-shaped subspace from X has been replaced with the (2,3,4) subspace from the indices. However, `x[:,ind_1,:,ind_2]` has shape (2,3,4,10,30,50) because there is no unambiguous place to drop in the indexing subspace, thus it is tacked-on to the beginning. It is always possible to use *`.transpose()`* to move the subspace anywhere desired. Note that this example cannot be replicated using *`take`*.

### Boolean array indexing

This advanced indexing occurs when obj is an array object of Boolean type, such as may be returned from comparison operators. A single boolean index array is practically identical to `x[obj.nonzero()]` where, as described above, *`obj.nonzero()`* returns a tuple (of length *`obj.ndim`*) of integer index arrays showing the `True` elements of *obj*. However, it is faster when `obj.shape == x.shape`.

If `obj.ndim == x.ndim`, `x[obj]` returns a 1-dimensional array filled with the elements of *x* corresponding to the `True` values of *obj*. The search order will be row-major, C-style. If *obj* has `True` values at entries that are outside of the bounds of *x*, then an index error will be raised. If *obj* is smaller than *x* it is identical to filling it with `False`.

**Example**

A common use case for this is filtering for desired element values. For example one may wish to select all entries from an array which are not NaN:

```
>>> x = np.array([[1., 2.], [np.nan, 3.], [np.nan, np.nan]])
>>> x[~np.isnan(x)]
array([1., 2., 3.])
```

Or wish to add a constant to all negative elements:

```
>>> x = np.array([1., -1., -2., 3])
>>> x[x < 0] += 20
>>> x
array([  1.,  19.,  18.,   3.])
```

---

In general if an index includes a Boolean array, the result will be identical to inserting `obj.nonzero()` into the same position and using the integer array indexing mechanism described above. `x[ind_1, boolean_array, ind_2]` is equivalent to `x[(ind_1,) + boolean_array.nonzero() + (ind_2,)]`.

If there is only one Boolean array and no integer indexing array present, this is straight forward. Care must only be taken to make sure that the boolean index has *exactly* as many dimensions as it is supposed to work with.

---

**Example**

From an array, select all rows which sum up to less or equal two:

```
>>> x = np.array([[0, 1], [1, 1], [2, 2]])
>>> rowsum = x.sum(-1)
>>> x[rowsum <= 2, :]
array([[0, 1],
       [1, 1]])
```

---

Combining multiple Boolean indexing arrays or a Boolean with an integer indexing array can best be understood with the *`obj.nonzero()`* analogy. The function *`ix_`* also supports boolean arrays and will work without any surprises.

---

**Example**

Use boolean indexing to select all rows adding up to an even number. At the same time columns 0 and 2 should be selected with an advanced integer index. Using the *`ix_`* function this can be done with:

```
>>> x = np.array([[ 0,  1,  2],
...               [ 3,  4,  5],
...               [ 6,  7,  8],
...               [ 9, 10, 11]])
>>> rows = (x.sum(-1) % 2) == 0
>>> rows
array([False,  True, False,  True])
>>> columns = [0, 2]
>>> x[np.ix_(rows, columns)]
array([[ 3,  5],
       [ 9, 11]])
```

Without the `np.ix_` call or only the diagonal elements would be selected.

Or without `np.ix_` (compare the integer array examples):

```
>>> rows = rows.nonzero()[0]
>>> x[rows[:, np.newaxis], columns]
```

(continues on next page)

(continued from previous page)

```
array([[ 3,  5],
       [ 9, 11]])
```

### 1.4.3 Detailed notes

These are some detailed notes, which are not of importance for day to day indexing (in no particular order):

- The native NumPy indexing type is `intp` and may differ from the default integer array type. `intp` is the smallest data type sufficient to safely index any array; for advanced indexing it may be faster than other types.
- For advanced assignments, there is in general no guarantee for the iteration order. This means that if an element is set more than once, it is not possible to predict the final result.
- An empty (tuple) index is a full scalar index into a zero dimensional array. `x[()]` returns a *scalar* if `x` is zero dimensional and a view otherwise. On the other hand `x[...]` always returns a view.
- If a zero dimensional array is present in the index *and* it is a full integer index the result will be a *scalar* and not a zero dimensional array. (Advanced indexing is not triggered.)
- When an ellipsis (`...`) is present but has no size (i.e. replaces zero `:`) the result will still always be an array. A view if no advanced index is present, otherwise a copy.
- the `nonzero` equivalence for Boolean arrays does not hold for zero dimensional boolean arrays.
- When the result of an advanced indexing operation has no elements but an individual index is out of bounds, whether or not an `IndexError` is raised is undefined (e.g. `x[[], [123]]` with `123` being out of bounds).
- When a *casting* error occurs during assignment (for example updating a numerical array using a sequence of strings), the array being assigned to may end up in an unpredictable partially updated state. However, if any other error (such as an out of bounds index) occurs, the array will remain unchanged.
- The memory layout of an advanced indexing result is optimized for each indexing operation and no particular memory order can be assumed.
- When using a subclass (especially one which manipulates its shape), the default `ndarray.__setitem__` behaviour will call `__getitem__` for *basic* indexing but not for *advanced* indexing. For such a subclass it may be preferable to call `ndarray.__setitem__` with a *base class* ndarray view on the data. This *must* be done if the subclasses `__getitem__` does not return views.

### 1.4.4 Field Access

**See also:**

*Data type objects (dtype)*, *Scalars*

If the *`ndarray`* object is a structured array the fields of the array can be accessed by indexing the array with strings, dictionary-like.

Indexing `x['field-name']` returns a new view to the array, which is of the same shape as *x* (except when the field is a sub-array) but of data type `x.dtype['field-name']` and contains only the part of the data in the specified field. Also *record array* scalars can be "indexed" this way.

Indexing into a structured array can also be done with a list of field names, *e.g.* `x[['field-name1', 'field-name2']]`. As of NumPy 1.16 this returns a view containing only those fields. In older versions of numpy it returned a copy. See the user guide section on structured_arrays for more information on multifield indexing.

If the accessed field is a sub-array, the dimensions of the sub-array are appended to the shape of the result.

---

**Example**

```
>>> x = np.zeros((2,2), dtype=[('a', np.int32), ('b', np.float64, (3,3))])
>>> x['a'].shape
(2, 2)
>>> x['a'].dtype
dtype('int32')
>>> x['b'].shape
(2, 2, 3, 3)
>>> x['b'].dtype
dtype('float64')
```

---

### 1.4.5 Flat Iterator indexing

`x.flat` returns an iterator that will iterate over the entire array (in C-contiguous style with the last index varying the fastest). This iterator object can also be indexed using basic slicing or advanced indexing as long as the selection object is not a tuple. This should be clear from the fact that `x.flat` is a 1-dimensional view. It can be used for integer indexing with 1-dimensional C-style-flat indices. The shape of any returned array is therefore the shape of the integer indexing object.

## 1.5 Iterating Over Arrays

The iterator object `nditer`, introduced in NumPy 1.6, provides many flexible ways to visit all the elements of one or more arrays in a systematic fashion. This page introduces some basic ways to use the object for computations on arrays in Python, then concludes with how one can accelerate the inner loop in Cython. Since the Python exposure of `nditer` is a relatively straightforward mapping of the C array iterator API, these ideas will also provide help working with array iteration from C or C++.

### 1.5.1 Single Array Iteration

The most basic task that can be done with the `nditer` is to visit every element of an array. Each element is provided one by one using the standard Python iterator interface.

---

**Example**

```
>>> a = np.arange(6).reshape(2,3)
>>> for x in np.nditer(a):
...     print(x, end=' ')
...
0 1 2 3 4 5
```

---

An important thing to be aware of for this iteration is that the order is chosen to match the memory layout of the array instead of using a standard C or Fortran ordering. This is done for access efficiency, reflecting the idea that by default one simply wants to visit each element without concern for a particular ordering. We can see this by iterating over the transpose of our previous array, compared to taking a copy of that transpose in C order.

---

**Example**

```
>>> a = np.arange(6).reshape(2,3)
>>> for x in np.nditer(a.T):
...     print(x, end=' ')
...
0 1 2 3 4 5
```

```
>>> for x in np.nditer(a.T.copy(order='C')):
...     print(x, end=' ')
...
0 3 1 4 2 5
```

---

The elements of both *a* and *a.T* get traversed in the same order, namely the order they are stored in memory, whereas the elements of *a.T.copy(order='C')* get visited in a different order because they have been put into a different memory layout.

### Controlling Iteration Order

There are times when it is important to visit the elements of an array in a specific order, irrespective of the layout of the elements in memory. The `nditer` object provides an *order* parameter to control this aspect of iteration. The default, having the behavior described above, is order='K' to keep the existing order. This can be overridden with order='C' for C order and order='F' for Fortran order.

---

**Example**

```
>>> a = np.arange(6).reshape(2,3)
>>> for x in np.nditer(a, order='F'):
...     print(x, end=' ')
...
0 3 1 4 2 5
>>> for x in np.nditer(a.T, order='C'):
...     print(x, end=' ')
...
0 3 1 4 2 5
```

---

### Modifying Array Values

By default, the `nditer` treats the input operand as a read-only object. To be able to modify the array elements, you must specify either read-write or write-only mode using the *'readwrite'* or *'writeonly'* per-operand flags.

The nditer will then yield writeable buffer arrays which you may modify. However, because the nditer must copy this buffer data back to the original array once iteration is finished, you must signal when the iteration is ended, by one of two methods. You may either:

- used the nditer as a context manager using the *with* statement, and the temporary data will be written back when the context is exited.
- call the iterator's *close* method once finished iterating, which will trigger the write-back.

The nditer can no longer be iterated once either *close* is called or its context is exited.

---

**Example**

```
>>> a = np.arange(6).reshape(2,3)
>>> a
array([[0, 1, 2],
       [3, 4, 5]])
>>> with np.nditer(a, op_flags=['readwrite']) as it:
...    for x in it:
...        x[...] = 2 * x
...
>>> a
array([[ 0,  2,  4],
       [ 6,  8, 10]])
```

---

If you are writing code that needs to support older versions of numpy, note that prior to 1.15, `nditer` was not a context manager and did not have a *close* method. Instead it relied on the destructor to initiate the writeback of the buffer.

### Using an External Loop

In all the examples so far, the elements of *a* are provided by the iterator one at a time, because all the looping logic is internal to the iterator. While this is simple and convenient, it is not very efficient. A better approach is to move the one-dimensional innermost loop into your code, external to the iterator. This way, NumPy's vectorized operations can be used on larger chunks of the elements being visited.

The `nditer` will try to provide chunks that are as large as possible to the inner loop. By forcing 'C' and 'F' order, we get different external loop sizes. This mode is enabled by specifying an iterator flag.

Observe that with the default of keeping native memory order, the iterator is able to provide a single one-dimensional chunk, whereas when forcing Fortran order, it has to provide three chunks of two elements each.

---

**Example**

```
>>> a = np.arange(6).reshape(2,3)
>>> for x in np.nditer(a, flags=['external_loop']):
...     print(x, end=' ')
...
[0 1 2 3 4 5]
```

```
>>> for x in np.nditer(a, flags=['external_loop'], order='F'):
...     print(x, end=' ')
...
[0 3] [1 4] [2 5]
```

---

### Tracking an Index or Multi-Index

During iteration, you may want to use the index of the current element in a computation. For example, you may want to visit the elements of an array in memory order, but use a C-order, Fortran-order, or multidimensional index to look up values in a different array.

The index is tracked by the iterator object itself, and accessible through the *index* or *multi_index* properties, depending on what was requested. The examples below show printouts demonstrating the progression of the index:

---

**Example**

```
>>> a = np.arange(6).reshape(2,3)
>>> it = np.nditer(a, flags=['f_index'])
>>> for x in it:
...     print("%d <%d>" % (x, it.index), end=' ')
...
0 <0> 1 <2> 2 <4> 3 <1> 4 <3> 5 <5>
```

```
>>> it = np.nditer(a, flags=['multi_index'])
>>> for x in it:
...     print("%d <%s>" % (x, it.multi_index), end=' ')
...
0 <(0, 0)> 1 <(0, 1)> 2 <(0, 2)> 3 <(1, 0)> 4 <(1, 1)> 5 <(1, 2)>
```

```
>>> with np.nditer(a, flags=['multi_index'], op_flags=['writeonly']) as it:
...     for x in it:
...         x[...] = it.multi_index[1] - it.multi_index[0]
...
>>> a
array([[ 0,  1,  2],
       [-1,  0,  1]])
```

---

Tracking an index or multi-index is incompatible with using an external loop, because it requires a different index value per element. If you try to combine these flags, the `nditer` object will raise an exception.

**Example**

```
>>> a = np.zeros((2,3))
>>> it = np.nditer(a, flags=['c_index', 'external_loop'])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: Iterator flag EXTERNAL_LOOP cannot be used if an index or multi-index is␣
↪being tracked
```

### Alternative Looping and Element Access

To make its properties more readily accessible during iteration, `nditer` has an alternative syntax for iterating, which works explicitly with the iterator object itself. With this looping construct, the current value is accessible by indexing into the iterator. Other properties, such as tracked indices remain as before. The examples below produce identical results to the ones in the previous section.

**Example**

```
>>> a = np.arange(6).reshape(2,3)
>>> it = np.nditer(a, flags=['f_index'])
>>> while not it.finished:
...     print("%d <%d>" % (it[0], it.index), end=' ')
...     is_not_finished = it.iternext()
...
0 <0> 1 <2> 2 <4> 3 <1> 4 <3> 5 <5>
```

```
>>> it = np.nditer(a, flags=['multi_index'])
>>> while not it.finished:
...     print("%d <%s>" % (it[0], it.multi_index), end=' ')
...     is_not_finished = it.iternext()
...
0 <(0, 0)> 1 <(0, 1)> 2 <(0, 2)> 3 <(1, 0)> 4 <(1, 1)> 5 <(1, 2)>
```

```
>>> with np.nditer(a, flags=['multi_index'], op_flags=['writeonly']) as it:
...     while not it.finished:
...         it[0] = it.multi_index[1] - it.multi_index[0]
...         is_not_finished = it.iternext()
...
>>> a
array([[ 0,  1,  2],
       [-1,  0,  1]])
```

---

### Buffering the Array Elements

When forcing an iteration order, we observed that the external loop option may provide the elements in smaller chunks because the elements can't be visited in the appropriate order with a constant stride. When writing C code, this is generally fine, however in pure Python code this can cause a significant reduction in performance.

By enabling buffering mode, the chunks provided by the iterator to the inner loop can be made larger, significantly reducing the overhead of the Python interpreter. In the example forcing Fortran iteration order, the inner loop gets to see all the elements in one go when buffering is enabled.

---

**Example**

```
>>> a = np.arange(6).reshape(2,3)
>>> for x in np.nditer(a, flags=['external_loop'], order='F'):
...     print(x, end=' ')
...
[0 3] [1 4] [2 5]
```

```
>>> for x in np.nditer(a, flags=['external_loop','buffered'], order='F'):
...     print(x, end=' ')
...
[0 3 1 4 2 5]
```

---

### Iterating as a Specific Data Type

There are times when it is necessary to treat an array as a different data type than it is stored as. For instance, one may want to do all computations on 64-bit floats, even if the arrays being manipulated are 32-bit floats. Except when writing low-level C code, it's generally better to let the iterator handle the copying or buffering instead of casting the data type yourself in the inner loop.

There are two mechanisms which allow this to be done, temporary copies and buffering mode. With temporary copies, a copy of the entire array is made with the new data type, then iteration is done in the copy. Write access is permitted through a mode which updates the original array after all the iteration is complete. The major drawback of temporary copies is that the temporary copy may consume a large amount of memory, particularly if the iteration data type has a larger itemsize than the original one.

Buffering mode mitigates the memory usage issue and is more cache-friendly than making temporary copies. Except for special cases, where the whole array is needed at once outside the iterator, buffering is recommended over temporary copying. Within NumPy, buffering is used by the ufuncs and other functions to support flexible inputs with minimal memory overhead.

In our examples, we will treat the input array with a complex data type, so that we can take square roots of negative numbers. Without enabling copies or buffering mode, the iterator will raise an exception if the data type doesn't match precisely.

**Example**

```
>>> a = np.arange(6).reshape(2,3) - 3
>>> for x in np.nditer(a, op_dtypes=['complex128']):
...     print(np.sqrt(x), end=' ')
...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Iterator operand required copying or buffering, but neither copying nor␣
↪buffering was enabled
```

In copying mode, 'copy' is specified as a per-operand flag. This is done to provide control in a per-operand fashion. Buffering mode is specified as an iterator flag.

**Example**

```
>>> a = np.arange(6).reshape(2,3) - 3
>>> for x in np.nditer(a, op_flags=['readonly','copy'],
...                 op_dtypes=['complex128']):
...     print(np.sqrt(x), end=' ')
...
1.7320508075688772j 1.4142135623730951j 1j 0j (1+0j) (1.4142135623730951+0j)
```

```
>>> for x in np.nditer(a, flags=['buffered'], op_dtypes=['complex128']):
...     print(np.sqrt(x), end=' ')
...
1.7320508075688772j 1.4142135623730951j 1j 0j (1+0j) (1.4142135623730951+0j)
```

The iterator uses NumPy's casting rules to determine whether a specific conversion is permitted. By default, it enforces 'safe' casting. This means, for example, that it will raise an exception if you try to treat a 64-bit float array as a 32-bit float array. In many cases, the rule 'same_kind' is the most reasonable rule to use, since it will allow conversion from 64 to 32-bit float, but not from float to int or from complex to float.

**Example**

```
>>> a = np.arange(6.)
>>> for x in np.nditer(a, flags=['buffered'], op_dtypes=['float32']):
...     print(x, end=' ')
...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Iterator operand 0 dtype could not be cast from dtype('float64') to dtype(
↪'float32') according to the rule 'safe'
```

```
>>> for x in np.nditer(a, flags=['buffered'], op_dtypes=['float32'],
...                 casting='same_kind'):
...     print(x, end=' ')
...
0.0 1.0 2.0 3.0 4.0 5.0
```

```
>>> for x in np.nditer(a, flags=['buffered'], op_dtypes=['int32'], casting='same_kind
↪'):
...     print(x, end=' ')
...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Iterator operand 0 dtype could not be cast from dtype('float64') to dtype(
↪'int32') according to the rule 'same_kind'
```

One thing to watch out for is conversions back to the original data type when using a read-write or write-only operand. A common case is to implement the inner loop in terms of 64-bit floats, and use 'same_kind' casting to allow the other floating-point types to be processed as well. While in read-only mode, an integer array could be provided, read-write mode will raise an exception because conversion back to the array would violate the casting rule.

**Example**

```
>>> a = np.arange(6)
>>> for x in np.nditer(a, flags=['buffered'], op_flags=['readwrite'],
...                 op_dtypes=['float64'], casting='same_kind'):
...     x[...] = x / 2.0
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
TypeError: Iterator requested dtype could not be cast from dtype('float64') to dtype(
↪'int64'), the operand 0 dtype, according to the rule 'same_kind'
```

## 1.5.2 Broadcasting Array Iteration

NumPy has a set of rules for dealing with arrays that have differing shapes which are applied whenever functions take multiple operands which combine element-wise. This is called *broadcasting*. The `nditer` object can apply these rules for you when you need to write such a function.

As an example, we print out the result of broadcasting a one and a two dimensional array together.

**Example**

```
>>> a = np.arange(3)
>>> b = np.arange(6).reshape(2,3)
>>> for x, y in np.nditer([a,b]):
...     print("%d:%d" % (x,y), end=' ')
...
0:0 1:1 2:2 0:3 1:4 2:5
```

When a broadcasting error occurs, the iterator raises an exception which includes the input shapes to help diagnose the problem.

---

**Example**

```
>>> a = np.arange(2)
>>> b = np.arange(6).reshape(2,3)
>>> for x, y in np.nditer([a,b]):
...     print("%d:%d" % (x,y), end=' ')
...
Traceback (most recent call last):
...
ValueError: operands could not be broadcast together with shapes (2,) (2,3)
```

---

### Iterator-Allocated Output Arrays

A common case in NumPy functions is to have outputs allocated based on the broadcasting of the input, and additionally have an optional parameter called 'out' where the result will be placed when it is provided. The *nditer* object provides a convenient idiom that makes it very easy to support this mechanism.

We'll show how this works by creating a function *square* which squares its input. Let's start with a minimal function definition excluding 'out' parameter support.

---

**Example**

```
>>> def square(a):
...     with np.nditer([a, None]) as it:
...         for x, y in it:
...             y[...] = x*x
...         return it.operands[1]
...
>>> square([1,2,3])
array([1, 4, 9])
```

---

By default, the *nditer* uses the flags 'allocate' and 'writeonly' for operands that are passed in as None. This means we were able to provide just the two operands to the iterator, and it handled the rest.

When adding the 'out' parameter, we have to explicitly provide those flags, because if someone passes in an array as 'out', the iterator will default to 'readonly', and our inner loop would fail. The reason 'readonly' is the default for input arrays is to prevent confusion about unintentionally triggering a reduction operation. If the default were 'readwrite', any broadcasting operation would also trigger a reduction, a topic which is covered later in this document.

While we're at it, let's also introduce the 'no_broadcast' flag, which will prevent the output from being broadcast. This is important, because we only want one input value for each output. Aggregating more than one input value is a reduction operation which requires special handling. It would already raise an error because reductions must be explicitly enabled in an iterator flag, but the error message that results from disabling broadcasting is much more understandable for end-users. To see how to generalize the square function to a reduction, look at the sum of squares function in the section about Cython.

For completeness, we'll also add the 'external_loop' and 'buffered' flags, as these are what you will typically want for performance reasons.

---

**Example**

```
>>> def square(a, out=None):
...     it = np.nditer([a, out],
...             flags = ['external_loop', 'buffered'],
...             op_flags = [['readonly'],
...                         ['writeonly', 'allocate', 'no_broadcast']])
...     with it:
...         for x, y in it:
...             y[...] = x*x
...         return it.operands[1]
...
```

```
>>> square([1,2,3])
array([1, 4, 9])
```

```
>>> b = np.zeros((3,))
>>> square([1,2,3], out=b)
array([ 1.,  4.,  9.])
>>> b
array([ 1.,  4.,  9.])
```

```
>>> square(np.arange(6).reshape(2,3), out=b)
Traceback (most recent call last):
  ...
ValueError: non-broadcastable output operand with shape (3,) doesn't
match the broadcast shape (2,3)
```

---

### Outer Product Iteration

Any binary operation can be extended to an array operation in an outer product fashion like in `outer`, and the `nditer` object provides a way to accomplish this by explicitly mapping the axes of the operands. It is also possible to do this with `newaxis` indexing, but we will show you how to directly use the nditer *op_axes* parameter to accomplish this with no intermediate views.

We'll do a simple outer product, placing the dimensions of the first operand before the dimensions of the second operand. The *op_axes* parameter needs one list of axes for each operand, and provides a mapping from the iterator's axes to the axes of the operand.

Suppose the first operand is one dimensional and the second operand is two dimensional. The iterator will have three dimensions, so *op_axes* will have two 3-element lists. The first list picks out the one axis of the first operand, and is -1 for the rest of the iterator axes, with a final result of [0, -1, -1]. The second list picks out the two axes of the second operand, but shouldn't overlap with the axes picked out in the first operand. Its list is [-1, 0, 1]. The output operand maps onto the iterator axes in the standard manner, so we can provide None instead of constructing another list.

The operation in the inner loop is a straightforward multiplication. Everything to do with the outer product is handled by the iterator setup.

---

**Example**

```
>>> a = np.arange(3)
>>> b = np.arange(8).reshape(2,4)
>>> it = np.nditer([a, b, None], flags=['external_loop'],
...             op_axes=[[0, -1, -1], [-1, 0, 1], None])
>>> with it:
```

(continues on next page)

(continued from previous page)

```
...     for x, y, z in it:
...         z[...] = x*y
...     result = it.operands[2]  # same as z
...
>>> result
array([[[ 0,  0,  0,  0],
        [ 0,  0,  0,  0]],
       [[ 0,  1,  2,  3],
        [ 4,  5,  6,  7]],
       [[ 0,  2,  4,  6],
        [ 8, 10, 12, 14]]])
```

Note that once the iterator is closed we can not access `operands` and must use a reference created inside the context manager.

### Reduction Iteration

Whenever a writeable operand has fewer elements than the full iteration space, that operand is undergoing a reduction. The `nditer` object requires that any reduction operand be flagged as read-write, and only allows reductions when 'reduce_ok' is provided as an iterator flag.

For a simple example, consider taking the sum of all elements in an array.

**Example**

```
>>> a = np.arange(24).reshape(2,3,4)
>>> b = np.array(0)
>>> with np.nditer([a, b], flags=['reduce_ok'],
...                     op_flags=[['readonly'], ['readwrite']]) as it:
...     for x,y in it:
...         y[...] += x
...
>>> b
array(276)
>>> np.sum(a)
276
```

Things are a little bit more tricky when combining reduction and allocated operands. Before iteration is started, any reduction operand must be initialized to its starting values. Here's how we can do this, taking sums along the last axis of *a*.

**Example**

```
>>> a = np.arange(24).reshape(2,3,4)
>>> it = np.nditer([a, None], flags=['reduce_ok'],
...             op_flags=[['readonly'], ['readwrite', 'allocate']],
...             op_axes=[None, [0,1,-1]])
>>> with it:
...     it.operands[1][...] = 0
...     for x, y in it:
...         y[...] += x
...     result = it.operands[1]
```

(continues on next page)

(continued from previous page)

```
...
>>> result
array([[ 6, 22, 38],
       [54, 70, 86]])
>>> np.sum(a, axis=2)
array([[ 6, 22, 38],
       [54, 70, 86]])
```

To do buffered reduction requires yet another adjustment during the setup. Normally the iterator construction involves copying the first buffer of data from the readable arrays into the buffer. Any reduction operand is readable, so it may be read into a buffer. Unfortunately, initialization of the operand after this buffering operation is complete will not be reflected in the buffer that the iteration starts with, and garbage results will be produced.

The iterator flag "delay_bufalloc" is there to allow iterator-allocated reduction operands to exist together with buffering. When this flag is set, the iterator will leave its buffers uninitialized until it receives a reset, after which it will be ready for regular iteration. Here's how the previous example looks if we also enable buffering.

**Example**

```
>>> a = np.arange(24).reshape(2,3,4)
>>> it = np.nditer([a, None], flags=['reduce_ok',
...                                  'buffered', 'delay_bufalloc'],
...                 op_flags=[['readonly'], ['readwrite', 'allocate']],
...                 op_axes=[None, [0,1,-1]])
>>> with it:
...     it.operands[1][...] = 0
...     it.reset()
...     for x, y in it:
...         y[...] += x
...     result = it.operands[1]
...
>>> result
array([[ 6, 22, 38],
       [54, 70, 86]])
```

## 1.5.3 Putting the Inner Loop in Cython

Those who want really good performance out of their low level operations should strongly consider directly using the iteration API provided in C, but for those who are not comfortable with C or C++, Cython is a good middle ground with reasonable performance tradeoffs. For the `nditer` object, this means letting the iterator take care of broadcasting, dtype conversion, and buffering, while giving the inner loop to Cython.

For our example, we'll create a sum of squares function. To start, let's implement this function in straightforward Python. We want to support an 'axis' parameter similar to the numpy `sum` function, so we will need to construct a list for the *op_axes* parameter. Here's how this looks.

**Example**

```
>>> def axis_to_axeslist(axis, ndim):
...     if axis is None:
...         return [-1] * ndim
```

(continues on next page)

(continued from previous page)

```
...     else:
...         if type(axis) is not tuple:
...             axis = (axis,)
...         axeslist = [1] * ndim
...         for i in axis:
...             axeslist[i] = -1
...         ax = 0
...         for i in range(ndim):
...             if axeslist[i] != -1:
...                 axeslist[i] = ax
...                 ax += 1
...         return axeslist
...
>>> def sum_squares_py(arr, axis=None, out=None):
...     axeslist = axis_to_axeslist(axis, arr.ndim)
...     it = np.nditer([arr, out], flags=['reduce_ok',
...                                       'buffered', 'delay_bufalloc'],
...                 op_flags=[['readonly'], ['readwrite', 'allocate']],
...                 op_axes=[None, axeslist],
...                 op_dtypes=['float64', 'float64'])
...     with it:
...         it.operands[1][...] = 0
...         it.reset()
...         for x, y in it:
...             y[...] += x*x
...         return it.operands[1]
...
>>> a = np.arange(6).reshape(2,3)
>>> sum_squares_py(a)
array(55.0)
>>> sum_squares_py(a, axis=-1)
array([  5.,  50.])
```

---

To Cython-ize this function, we replace the inner loop (y[…] += x*x) with Cython code that's specialized for the float64 dtype. With the 'external_loop' flag enabled, the arrays provided to the inner loop will always be one-dimensional, so very little checking needs to be done.

Here's the listing of sum_squares.pyx:

```
import numpy as np
cimport numpy as np
cimport cython

def axis_to_axeslist(axis, ndim):
    if axis is None:
        return [-1] * ndim
    else:
        if type(axis) is not tuple:
            axis = (axis,)
        axeslist = [1] * ndim
        for i in axis:
            axeslist[i] = -1
        ax = 0
        for i in range(ndim):
            if axeslist[i] != -1:
                axeslist[i] = ax
```

(continues on next page)

(continued from previous page)

```
                ax += 1
        return axeslist

@cython.boundscheck(False)
def sum_squares_cy(arr, axis=None, out=None):
    cdef np.ndarray[double] x
    cdef np.ndarray[double] y
    cdef int size
    cdef double value

    axeslist = axis_to_axeslist(axis, arr.ndim)
    it = np.nditer([arr, out], flags=['reduce_ok', 'external_loop',
                                      'buffered', 'delay_bufalloc'],
                op_flags=[['readonly'], ['readwrite', 'allocate']],
                op_axes=[None, axeslist],
                op_dtypes=['float64', 'float64'])
    with it:
        it.operands[1][...] = 0
        it.reset()
        for xarr, yarr in it:
            x = xarr
            y = yarr
            size = x.shape[0]
            for i in range(size):
               value = x[i]
               y[i] = y[i] + value * value
        return it.operands[1]
```

On this machine, building the .pyx file into a module looked like the following, but you may have to find some Cython tutorials to tell you the specifics for your system configuration.:

```
$ cython sum_squares.pyx
$ gcc -shared -pthread -fPIC -fwrapv -O2 -Wall -I/usr/include/python2.7 -fno-strict-
↪aliasing -o sum_squares.so sum_squares.c
```

Running this from the Python interpreter produces the same answers as our native Python/NumPy code did.

---

**Example**

```
>>> from sum_squares import sum_squares_cy
>>> a = np.arange(6).reshape(2,3)
>>> sum_squares_cy(a)
array(55.0)
>>> sum_squares_cy(a, axis=-1)
array([  5.,  50.])
```

---

Doing a little timing in IPython shows that the reduced overhead and memory allocation of the Cython inner loop is providing a very nice speedup over both the straightforward Python code and an expression using NumPy's built-in sum function.:

```
>>> a = np.random.rand(1000,1000)

>>> timeit sum_squares_py(a, axis=-1)
10 loops, best of 3: 37.1 ms per loop
```

(continues on next page)

(continued from previous page)

```
>>> timeit np.sum(a*a, axis=-1)
10 loops, best of 3: 20.9 ms per loop

>>> timeit sum_squares_cy(a, axis=-1)
100 loops, best of 3: 11.8 ms per loop

>>> np.all(sum_squares_cy(a, axis=-1) == np.sum(a*a, axis=-1))
True

>>> np.all(sum_squares_py(a, axis=-1) == np.sum(a*a, axis=-1))
True
```

## 1.6 Standard array subclasses

**Note:** Subclassing a `numpy.ndarray` is possible but if your goal is to create an array with *modified* behavior, as do dask arrays for distributed computation and cupy arrays for GPU-based computation, subclassing is discouraged. Instead, using numpy's dispatch mechanism is recommended.

The `ndarray` can be inherited from (in Python or in C) if desired. Therefore, it can form a foundation for many useful classes. Often whether to sub-class the array object or to simply use the core array component as an internal part of a new class is a difficult decision, and can be simply a matter of choice. NumPy has several tools for simplifying how your new object interacts with other array objects, and so the choice may not be significant in the end. One way to simplify the question is by asking yourself if the object you are interested in can be replaced as a single array or does it really require two or more arrays at its core.

Note that `asarray` always returns the base-class ndarray. If you are confident that your use of the array object can handle any subclass of an ndarray, then `asanyarray` can be used to allow subclasses to propagate more cleanly through your subroutine. In principal a subclass could redefine any aspect of the array and therefore, under strict guidelines, `asanyarray` would rarely be useful. However, most subclasses of the array object will not redefine certain aspects of the array object such as the buffer interface, or the attributes of the array. One important example, however, of why your subroutine may not be able to handle an arbitrary subclass of an array is that matrices redefine the "*" operator to be matrix-multiplication, rather than element-by-element multiplication.

### 1.6.1 Special attributes and methods

**See also:**

Subclassing ndarray

NumPy provides several hooks that classes can customize:

`class.`**`__array_ufunc__`**(*ufunc*, *method*, **inputs*, ***kwargs*)

New in version 1.13.

Any class, ndarray subclass or not, can define this method or set it to None in order to override the behavior of NumPy's ufuncs. This works quite similarly to Python's `__mul__` and other binary operation routines.

- *ufunc* is the ufunc object that was called.
- *method* is a string indicating which Ufunc method was called (one of `"__call__"`, `"reduce"`, `"reduceat"`, `"accumulate"`, `"outer"`, `"inner"`).

- *inputs* is a tuple of the input arguments to the `ufunc`.
- *kwargs* is a dictionary containing the optional input arguments of the ufunc. If given, any `out` arguments, both positional and keyword, are passed as a `tuple` in *kwargs*. See the discussion in *Universal functions (ufunc)* for details.

The method should return either the result of the operation, or `NotImplemented` if the operation requested is not implemented.

If one of the input or output arguments has a `__array_ufunc__` method, it is executed *instead* of the ufunc. If more than one of the arguments implements `__array_ufunc__`, they are tried in the order: subclasses before superclasses, inputs before outputs, otherwise left to right. The first routine returning something other than `NotImplemented` determines the result. If all of the `__array_ufunc__` operations return `NotImplemented`, a `TypeError` is raised.

---

**Note:** We intend to re-implement numpy functions as (generalized) Ufunc, in which case it will become possible for them to be overridden by the `__array_ufunc__` method. A prime candidate is `matmul`, which currently is not a Ufunc, but could be relatively easily be rewritten as a (set of) generalized Ufuncs. The same may happen with functions such as `median`, `amin`, and `argsort`.

---

Like with some other special methods in python, such as `__hash__` and `__iter__`, it is possible to indicate that your class does *not* support ufuncs by setting `__array_ufunc__ = None`. Ufuncs always raise `TypeError` when called on an object that sets `__array_ufunc__ = None`.

The presence of `__array_ufunc__` also influences how `ndarray` handles binary operations like `arr + obj` and `arr < obj` when `arr` is an `ndarray` and `obj` is an instance of a custom class. There are two possibilities. If `obj.__array_ufunc__` is present and not None, then `ndarray.__add__` and friends will delegate to the ufunc machinery, meaning that `arr + obj` becomes `np.add(arr, obj)`, and then `add` invokes `obj.__array_ufunc__`. This is useful if you want to define an object that acts like an array.

Alternatively, if `obj.__array_ufunc__` is set to None, then as a special case, special methods like `ndarray.__add__` will notice this and *unconditionally* raise `TypeError`. This is useful if you want to create objects that interact with arrays via binary operations, but are not themselves arrays. For example, a units handling system might have an object `m` representing the "meters" unit, and want to support the syntax `arr * m` to represent that the array has units of "meters", but not want to otherwise interact with arrays via ufuncs or otherwise. This can be done by setting `__array_ufunc__ = None` and defining `__mul__` and `__rmul__` methods. (Note that this means that writing an `__array_ufunc__` that always returns `NotImplemented` is not quite the same as setting `__array_ufunc__ = None`: in the former case, `arr + obj` will raise `TypeError`, while in the latter case it is possible to define a `__radd__` method to prevent this.)

The above does not hold for in-place operators, for which `ndarray` never returns `NotImplemented`. Hence, `arr += obj` would always lead to a `TypeError`. This is because for arrays in-place operations cannot generically be replaced by a simple reverse operation. (For instance, by default, `arr += obj` would be translated to `arr = arr + obj`, i.e., `arr` would be replaced, contrary to what is expected for in-place array operations.)

---

**Note:** If you define `__array_ufunc__`:

- If you are not a subclass of `ndarray`, we recommend your class define special methods like `__add__` and `__lt__` that delegate to ufuncs just like ndarray does. An easy way to do this is to subclass from `NDArrayOperatorsMixin`.
- If you subclass `ndarray`, we recommend that you put all your override logic in `__array_ufunc__` and not also override special methods. This ensures the class hierarchy is determined in only one place rather than separately by the ufunc machinery and by the binary operation rules (which gives preference to special methods of subclasses; the alternative way to enforce a one-place only hierarchy, of setting `__array_ufunc__` to None, would seem very unexpected and thus confusing, as then the subclass would not work at all with ufuncs).

- *`ndarray`* defines its own *`__array_ufunc__`*, which, evaluates the ufunc if no arguments have overrides, and returns `NotImplemented` otherwise. This may be useful for subclasses for which *`__array_ufunc__`* converts any instances of its own class to *`ndarray`*: it can then pass these on to its superclass using `super().__array_ufunc__(*inputs, **kwargs)`, and finally return the results after possible back-conversion. The advantage of this practice is that it ensures that it is possible to have a hierarchy of subclasses that extend the behaviour. See Subclassing ndarray for details.

---

**Note:** If a class defines the *`__array_ufunc__`* method, this disables the *`__array_wrap__`*, *`__array_prepare__`*, *`__array_priority__`* mechanism described below for ufuncs (which may eventually be deprecated).

---

`class.`**`__array_function__`**(*func*, *types*, *args*, *kwargs*)

New in version 1.16.

---

**Note:**

- In NumPy 1.17, the protocol is enabled by default, but can be disabled with `NUMPY_EXPERIMENTAL_ARRAY_FUNCTION=0`.
- In NumPy 1.16, you need to set the environment variable `NUMPY_EXPERIMENTAL_ARRAY_FUNCTION=1` before importing NumPy to use NumPy function overrides.
- Eventually, expect to `__array_function__` to always be enabled.

---

- `func` is an arbitrary callable exposed by NumPy's public API, which was called in the form `func(*args, **kwargs)`.
- `types` is a collection of unique argument types from the original NumPy function call that implement `__array_function__`.
- The tuple `args` and dict `kwargs` are directly passed on from the original call.

As a convenience for `__array_function__` implementors, `types` provides all argument types with an `'__array_function__'` attribute. This allows implementors to quickly identify cases where they should defer to `__array_function__` implementations on other arguments. Implementations should not rely on the iteration order of `types`.

Most implementations of `__array_function__` will start with two checks:

1. Is the given function something that we know how to overload?
2. Are all arguments of a type that we know how to handle?

If these conditions hold, `__array_function__` should return the result from calling its implementation for `func(*args, **kwargs)`. Otherwise, it should return the sentinel value `NotImplemented`, indicating that the function is not implemented by these types.

There are no general requirements on the return value from `__array_function__`, although most sensible implementations should probably return array(s) with the same type as one of the function's arguments.

It may also be convenient to define a custom decorators (`implements` below) for registering `__array_function__` implementations.

```
HANDLED_FUNCTIONS = {}

```

(continues on next page)

(continued from previous page)

```
class MyArray:
    def __array_function__(self, func, types, args, kwargs):
        if func not in HANDLED_FUNCTIONS:
            return NotImplemented
        # Note: this allows subclasses that don't override
        # __array_function__ to handle MyArray objects
        if not all(issubclass(t, MyArray) for t in types):
            return NotImplemented
        return HANDLED_FUNCTIONS[func](*args, **kwargs)

def implements(numpy_function):
    """Register an __array_function__ implementation for MyArray objects."""
    def decorator(func):
        HANDLED_FUNCTIONS[numpy_function] = func
        return func
    return decorator

@implements(np.concatenate)
def concatenate(arrays, axis=0, out=None):
    ...  # implementation of concatenate for MyArray objects

@implements(np.broadcast_to)
def broadcast_to(array, shape):
    ...  # implementation of broadcast_to for MyArray objects
```

Note that it is not required for `__array_function__` implementations to include *all* of the corresponding NumPy function's optional arguments (e.g., `broadcast_to` above omits the irrelevant `subok` argument). Optional arguments are only passed in to `__array_function__` if they were explicitly used in the NumPy function call.

Just like the case for builtin special methods like `__add__`, properly written `__array_function__` methods should always return `NotImplemented` when an unknown type is encountered. Otherwise, it will be impossible to correctly override NumPy functions from another object if the operation also includes one of your objects.

For the most part, the rules for dispatch with `__array_function__` match those for `__array_ufunc__`. In particular:

- NumPy will gather implementations of `__array_function__` from all specified inputs and call them in order: subclasses before superclasses, and otherwise left to right. Note that in some edge cases involving subclasses, this differs slightly from the current behavior of Python.
- Implementations of `__array_function__` indicate that they can handle the operation by returning any value other than `NotImplemented`.
- If all `__array_function__` methods return `NotImplemented`, NumPy will raise `TypeError`.

If no `__array_function__` methods exists, NumPy will default to calling its own implementation, intended for use on NumPy arrays. This case arises, for example, when all array-like arguments are Python numbers or lists. (NumPy arrays do have a `__array_function__` method, given below, but it always returns `NotImplemented` if any argument other than a NumPy array subclass implements `__array_function__`.)

One deviation from the current behavior of `__array_ufunc__` is that NumPy will only call `__array_function__` on the *first* argument of each unique type. This matches Python's rule for calling reflected methods, and this ensures that checking overloads has acceptable performance even when there are a large number of overloaded arguments.

`class.`**`__array_finalize__`**(*obj*)

This method is called whenever the system internally allocates a new array from *obj*, where *obj* is a subclass (subtype) of the `ndarray`. It can be used to change attributes of *self* after construction (so as to ensure a 2-d matrix for example), or to update meta-information from the "parent." Subclasses inherit a default implementation of this method that does nothing.

`class.`**`__array_prepare__`**(*array*, *context=None*)

At the beginning of every *ufunc*, this method is called on the input object with the highest array priority, or the output object if one was specified. The output array is passed in and whatever is returned is passed to the ufunc. Subclasses inherit a default implementation of this method which simply returns the output array unmodified. Subclasses may opt to use this method to transform the output array into an instance of the subclass and update metadata before returning the array to the ufunc for computation.

**Note:** For ufuncs, it is hoped to eventually deprecate this method in favour of `__array_ufunc__`.

`class.`**`__array_wrap__`**(*array*, *context=None*)

At the end of every *ufunc*, this method is called on the input object with the highest array priority, or the output object if one was specified. The ufunc-computed array is passed in and whatever is returned is passed to the user. Subclasses inherit a default implementation of this method, which transforms the array into a new instance of the object's class. Subclasses may opt to use this method to transform the output array into an instance of the subclass and update metadata before returning the array to the user.

**Note:** For ufuncs, it is hoped to eventually deprecate this method in favour of `__array_ufunc__`.

`class.`**`__array_priority__`**

The value of this attribute is used to determine what type of object to return in situations where there is more than one possibility for the Python type of the returned object. Subclasses inherit a default value of 0.0 for this attribute.

**Note:** For ufuncs, it is hoped to eventually deprecate this method in favour of `__array_ufunc__`.

`class.`**`__array__`**([*dtype*])

If a class (ndarray subclass or not) having the `__array__` method is used as the output object of an *ufunc*, results will be written to the object returned by `__array__`. Similar conversion is done on input arrays.

### 1.6.2 Matrix objects

**Note:** It is strongly advised *not* to use the matrix subclass. As described below, it makes writing functions that deal consistently with matrices and regular arrays very difficult. Currently, they are mainly used for interacting with `scipy.sparse`. We hope to provide an alternative for this use, however, and eventually remove the `matrix` subclass.

`matrix` objects inherit from the ndarray and therefore, they have the same attributes and methods of ndarrays. There are six important differences of matrix objects, however, that may lead to unexpected results when you use matrices but expect them to act like arrays:

1. Matrix objects can be created using a string notation to allow Matlab-style syntax where spaces separate columns and semicolons (';') separate rows.
2. Matrix objects are always two-dimensional. This has far-reaching implications, in that m.ravel() is still two-dimensional (with a 1 in the first dimension) and item selection returns two-dimensional objects so that sequence behavior is fundamentally different than arrays.
3. Matrix objects over-ride multiplication to be matrix-multiplication. **Make sure you understand this for functions that you may want to receive matrices. Especially in light of the fact that asanyarray(m) returns a matrix**

**when m is a matrix.**

4. Matrix objects over-ride power to be matrix raised to a power. The same warning about using power inside a function that uses asanyarray(…) to get an array object holds for this fact.
5. The default __array_priority__ of matrix objects is 10.0, and therefore mixed operations with ndarrays always produce matrices.
6. Matrices have special attributes which make calculations easier. These are

| | |
|---|---|
| `matrix.T` | Returns the transpose of the matrix. |
| `matrix.H` | Returns the (complex) conjugate transpose of *self*. |
| `matrix.I` | Returns the (multiplicative) inverse of invertible *self*. |
| `matrix.A` | Return *self* as an `ndarray` object. |

property

**property** matrix.**T**

Returns the transpose of the matrix.

Does *not* conjugate! For the complex conjugate transpose, use `.H`.

**Parameters**

**None**

**Returns**

**ret**

[matrix object] The (non-conjugated) transpose of the matrix.

**See also:**

`transpose`, `getH`

### Examples

```
>>> m = np.matrix('[1, 2; 3, 4]')
>>> m
matrix([[1, 2],
        [3, 4]])
>>> m.getT()
matrix([[1, 3],
        [2, 4]])
```

property

**property** matrix.**H**

Returns the (complex) conjugate transpose of *self*.

Equivalent to `np.transpose(self)` if *self* is real-valued.

**Parameters**

**None**

**Returns**

**ret**

[matrix object] complex conjugate transpose of *self*

### Examples

```
>>> x = np.matrix(np.arange(12).reshape((3,4)))
>>> z = x - 1j*x; z
matrix([[  0. +0.j,   1. -1.j,   2. -2.j,   3. -3.j],
        [  4. -4.j,   5. -5.j,   6. -6.j,   7. -7.j],
        [  8. -8.j,   9. -9.j,  10.-10.j,  11.-11.j]])
>>> z.getH()
matrix([[ 0. -0.j,  4. +4.j,  8. +8.j],
        [ 1. +1.j,  5. +5.j,  9. +9.j],
        [ 2. +2.j,  6. +6.j, 10.+10.j],
        [ 3. +3.j,  7. +7.j, 11.+11.j]])
```

property

**property** `matrix.`**`I`**

Returns the (multiplicative) inverse of invertible *self*.

**Parameters**

**None**

**Returns**

**ret**

[matrix object] If *self* is non-singular, *ret* is such that `ret * self` == `self * ret` == `np.matrix(np.eye(self[0,:].size))` all return `True`.

**Raises**

**numpy.linalg.LinAlgError: Singular matrix**

If *self* is singular.

**See also:**

*`linalg.inv`*

### Examples

```
>>> m = np.matrix('[1, 2; 3, 4]'); m
matrix([[1, 2],
        [3, 4]])
>>> m.getI()
matrix([[-2. ,  1. ],
        [ 1.5, -0.5]])
>>> m.getI() * m
matrix([[ 1.,  0.], # may vary
        [ 0.,  1.]])
```

property

**property** `matrix.`**A**

Return *self* as an `ndarray` object.

Equivalent to `np.asarray(self)`.

**Parameters**

**None**

**Returns**

**ret**

[ndarray] *self* as an `ndarray`

## Examples

```
>>> x = np.matrix(np.arange(12).reshape((3,4))); x
matrix([[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [ 8,  9, 10, 11]])
>>> x.getA()
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

**Warning:** Matrix objects over-ride multiplication, '*', and power, '**', to be matrix-multiplication and matrix power, respectively. If your subroutine can accept sub-classes and you do not convert to base- class arrays, then you must use the ufuncs multiply and power to be sure that you are performing the correct operation for all inputs.

The matrix class is a Python subclass of the ndarray and can be used as a reference for how to construct your own subclass of the ndarray. Matrices can be created from other matrices, strings, and anything else that can be converted to an `ndarray` . The name "mat "is an alias for "matrix "in NumPy.

| | |
|---|---|
| *`matrix`*(data[, dtype, copy]) | **Note:** It is no longer recommended to use this class, even for linear |
| *`asmatrix`*(data[, dtype]) | Interpret the input as a matrix. |
| *`bmat`*(obj[, ldict, gdict]) | Build a matrix object from a string, nested sequence, or array. |

**class** `numpy.`**`matrix`**(*data*, *dtype=None*, *copy=True*)

**Note:** It is no longer recommended to use this class, even for linear algebra. Instead use regular arrays. The class may be removed in the future.

Returns a matrix from an array-like object, or from a string of data. A matrix is a specialized 2-D array that retains its 2-D nature through operations. It has certain special operators, such as `*` (matrix multiplication) and `**` (matrix power).

**Parameters**

**data**

[array_like or string] If `data` is a string, it is interpreted as a matrix with commas or spaces separating columns, and semicolons separating rows.

**dtype**

[data-type] Data-type of the output matrix.

**copy**

[bool] If `data` is already an *`ndarray`*, then this flag determines whether the data is copied (the default), or whether a view is constructed.

**See also:**

*`array`*

## Examples

```
>>> a = np.matrix('1 2; 3 4')
>>> a
matrix([[1, 2],
        [3, 4]])
```

```
>>> np.matrix([[1, 2], [3, 4]])
matrix([[1, 2],
        [3, 4]])
```

**Attributes**

***`A`***

Return *self* as an *`ndarray`* object.

**`A1`**

Return *self* as a flattened *`ndarray`*.

***`H`***

Returns the (complex) conjugate transpose of *self*.

***`I`***

Returns the (multiplicative) inverse of invertible *self*.

***`T`***

Returns the transpose of the matrix.

**`base`**

Base object if memory is from some other object.

**`ctypes`**

An object to simplify the interaction of the array with the ctypes module.

**`data`**

Python buffer object pointing to the start of the array's data.

***`dtype`***

Data-type of the array's elements.

**`flags`**

Information about the memory layout of the array.

**`flat`**

A 1-D iterator over the array.

***`imag`***

The imaginary part of the array.

**`itemsize`**

Length of one array element in bytes.

**`nbytes`**

Total bytes consumed by the elements of the array.

**`ndim`**

Number of array dimensions.

***`real`***

The real part of the array.

***`shape`***

Tuple of array dimensions.

**`size`**

Number of elements in the array.

**`strides`**

Tuple of bytes to step in each dimension when traversing an array.

## Methods

| | |
|---|---|
| `all`(self[, axis, out]) | Test whether all matrix elements along a given axis evaluate to True. |
| `any`(self[, axis, out]) | Test whether any array element along a given axis evaluates to True. |
| `argmax`(self[, axis, out]) | Indexes of the maximum values along an axis. |
| `argmin`(self[, axis, out]) | Indexes of the minimum values along an axis. |
| `argpartition`(kth[, axis, kind, order]) | Returns the indices that would partition this array. |
| `argsort`([axis, kind, order]) | Returns the indices that would sort this array. |
| `astype`(dtype[, order, casting, subok, copy]) | Copy of the array, cast to a specified type. |
| `byteswap`([inplace]) | Swap the bytes of the array elements |
| `choose`(choices[, out, mode]) | Use an index array to construct a new array from a set of choices. |

Continued on next page

Table 37 – continued from previous page

| | |
|---|---|
| `clip`([min, max, out]) | Return an array whose values are limited to `[min, max]`. |
| `compress`(condition[, axis, out]) | Return selected slices of this array along given axis. |
| `conj`() | Complex-conjugate all elements. |
| `conjugate`() | Return the complex conjugate, element-wise. |
| `copy`([order]) | Return a copy of the array. |
| `cumprod`([axis, dtype, out]) | Return the cumulative product of the elements along the given axis. |
| `cumsum`([axis, dtype, out]) | Return the cumulative sum of the elements along the given axis. |
| `diagonal`([offset, axis1, axis2]) | Return specified diagonals. |
| `dot`(b[, out]) | Dot product of two arrays. |
| `dump`(file) | Dump a pickle of the array to the specified file. |
| `dumps`() | Returns the pickle of the array as a string. |
| `fill`(value) | Fill the array with a scalar value. |
| `flatten`(self[, order]) | Return a flattened copy of the matrix. |
| `getA`(self) | Return *self* as an `ndarray` object. |
| `getA1`(self) | Return *self* as a flattened `ndarray`. |
| `getH`(self) | Returns the (complex) conjugate transpose of *self*. |
| `getI`(self) | Returns the (multiplicative) inverse of invertible *self*. |
| `getT`(self) | Returns the transpose of the matrix. |
| `getfield`(dtype[, offset]) | Returns a field of the given array as a certain type. |
| `item`(*args) | Copy an element of an array to a standard Python scalar and return it. |
| `itemset`(*args) | Insert scalar into an array (scalar is cast to array's dtype, if possible) |
| `max`(self[, axis, out]) | Return the maximum value along an axis. |
| `mean`(self[, axis, dtype, out]) | Returns the average of the matrix elements along the given axis. |
| `min`(self[, axis, out]) | Return the minimum value along an axis. |
| `newbyteorder`([new_order]) | Return the array with the same data viewed with a different byte order. |
| `nonzero`() | Return the indices of the elements that are non-zero. |
| `partition`(kth[, axis, kind, order]) | Rearranges the elements in the array in such a way that the value of the element in kth position is in the position it would be in a sorted array. |
| `prod`(self[, axis, dtype, out]) | Return the product of the array elements over the given axis. |
| `ptp`(self[, axis, out]) | Peak-to-peak (maximum - minimum) value along the given axis. |
| `put`(indices, values[, mode]) | Set `a.flat[n] = values[n]` for all *n* in indices. |
| `ravel`(self[, order]) | Return a flattened matrix. |
| `repeat`(repeats[, axis]) | Repeat elements of an array. |
| `reshape`(shape[, order]) | Returns an array containing the same data with a new shape. |
| `resize`(new_shape[, refcheck]) | Change shape and size of array in-place. |
| `round`([decimals, out]) | Return *a* with each element rounded to the given number of decimals. |
| `searchsorted`(v[, side, sorter]) | Find indices where elements of v should be inserted in a to maintain order. |

Continued on next page

Table 37 – continued from previous page

| | |
|---|---|
| `setfield`(val, dtype[, offset]) | Put a value into a specified place in a field defined by a data-type. |
| `setflags`([write, align, uic]) | Set array flags WRITEABLE, ALIGNED, (WRITEBACKIFCOPY and UPDATEIFCOPY), respectively. |
| `sort`([axis, kind, order]) | Sort an array in-place. |
| `squeeze`(self[, axis]) | Return a possibly reshaped matrix. |
| `std`(self[, axis, dtype, out, ddof]) | Return the standard deviation of the array elements along the given axis. |
| `sum`(self[, axis, dtype, out]) | Returns the sum of the matrix elements, along the given axis. |
| `swapaxes`(axis1, axis2) | Return a view of the array with *axis1* and *axis2* interchanged. |
| `take`(indices[, axis, out, mode]) | Return an array formed from the elements of *a* at the given indices. |
| `tobytes`([order]) | Construct Python bytes containing the raw data bytes in the array. |
| `tofile`(fid[, sep, format]) | Write array to a file as text or binary (default). |
| `tolist`(self) | Return the matrix as a (possibly nested) list. |
| `tostring`([order]) | A compatibility alias for `tobytes`, with exactly the same behavior. |
| `trace`([offset, axis1, axis2, dtype, out]) | Return the sum along diagonals of the array. |
| `transpose`(*axes) | Returns a view of the array with axes transposed. |
| `var`(self[, axis, dtype, out, ddof]) | Returns the variance of the matrix elements, along the given axis. |
| `view`([dtype][, type]) | New view of array with the same data. |

method

`matrix.`**`all`**(*self*, *axis=None*, *out=None*)

Test whether all matrix elements along a given axis evaluate to True.

**Parameters**

**See ‘numpy.all‘ for complete descriptions**

**See also:**

`numpy.all`

### Notes

This is the same as `ndarray.all`, but it returns a `matrix` object.

### Examples

```
>>> x = np.matrix(np.arange(12).reshape((3,4))); x
matrix([[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [ 8,  9, 10, 11]])
>>> y = x[0]; y
matrix([[0, 1, 2, 3]])
>>> (x == y)
matrix([[ True,  True,  True,  True],
        [False, False, False, False],
        [False, False, False, False]])
>>> (x == y).all()
False
>>> (x == y).all(0)
matrix([[False, False, False, False]])
>>> (x == y).all(1)
matrix([[ True],
        [False],
        [False]])
```

method

`matrix.`**`any`**(*self*, *axis=None*, *out=None*)

Test whether any array element along a given axis evaluates to True.

Refer to `numpy.any` for full documentation.

**Parameters**

**axis**

[int, optional] Axis along which logical OR is performed

**out**

[ndarray, optional] Output to existing array instead of creating new one, must have same shape as expected output

**Returns**

**any**

[bool, ndarray] Returns a single bool if *axis* is `None`; otherwise, returns `ndarray`

method

`matrix.`**`argmax`**(*self*, *axis=None*, *out=None*)

Indexes of the maximum values along an axis.

Return the indexes of the first occurrences of the maximum values along the specified axis. If axis is None, the index is for the flattened matrix.

**Parameters**

**See 'numpy.argmax' for complete descriptions**

**See also:**

*numpy.argmax*

### Notes

This is the same as `ndarray.argmax`, but returns a `matrix` object where `ndarray.argmax` would return an `ndarray`.

### Examples

```
>>> x = np.matrix(np.arange(12).reshape((3,4))); x
matrix([[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [ 8,  9, 10, 11]])
>>> x.argmax()
11
>>> x.argmax(0)
matrix([[2, 2, 2, 2]])
>>> x.argmax(1)
matrix([[3],
        [3],
        [3]])
```

method

`matrix.`**`argmin`**(*self*, *axis=None*, *out=None*)

Indexes of the minimum values along an axis.

Return the indexes of the first occurrences of the minimum values along the specified axis. If axis is None, the index is for the flattened matrix.

**Parameters**

**See 'numpy.argmin' for complete descriptions.**

**See also:**

*numpy.argmin*

### Notes

This is the same as `ndarray.argmin`, but returns a `matrix` object where `ndarray.argmin` would return an `ndarray`.

### Examples

```
>>> x = -np.matrix(np.arange(12).reshape((3,4))); x
matrix([[  0,  -1,  -2,  -3],
        [ -4,  -5,  -6,  -7],
        [ -8,  -9, -10, -11]])
>>> x.argmin()
11
>>> x.argmin(0)
matrix([[2, 2, 2, 2]])
>>> x.argmin(1)
matrix([[3],
        [3],
        [3]])
```

method

`matrix.`**`argpartition`**(*kth*, *axis=-1*, *kind='introselect'*, *order=None*)

Returns the indices that would partition this array.

Refer to *`numpy.argpartition`* for full documentation.

New in version 1.8.0.

**See also:**

***`numpy.argpartition`***

equivalent function

method

`matrix.`**`argsort`**(*axis=-1*, *kind=None*, *order=None*)

Returns the indices that would sort this array.

Refer to *`numpy.argsort`* for full documentation.

**See also:**

***`numpy.argsort`***

equivalent function

method

`matrix.`**`astype`**(*dtype*, *order='K'*, *casting='unsafe'*, *subok=True*, *copy=True*)

Copy of the array, cast to a specified type.

**Parameters**

**dtype**

[str or dtype] Typecode or data-type to which the array is cast.

**order**

[{'C', 'F', 'A', 'K'}, optional] Controls the memory layout order of the result. 'C' means C order, 'F' means Fortran order, 'A' means 'F' order if all the arrays are Fortran contiguous, 'C' order otherwise, and 'K' means as close to the order the array elements appear in memory as possible. Default is 'K'.

**casting**

[{‘no’, ‘equiv’, ‘safe’, ‘same_kind’, ‘unsafe’}, optional] Controls what kind of data casting may occur. Defaults to ‘unsafe’ for backwards compatibility.

- ‘no’ means the data types should not be cast at all.
- ‘equiv’ means only byte-order changes are allowed.
- ‘safe’ means only casts which can preserve values are allowed.
- ‘same_kind’ means only safe casts or casts within a kind, like float64 to float32, are allowed.
- ‘unsafe’ means any data conversions may be done.

**subok**

[bool, optional] If True, then sub-classes will be passed-through (default), otherwise the returned array will be forced to be a base-class array.

**copy**

[bool, optional] By default, astype always returns a newly allocated array. If this is set to false, and the `dtype`, *order*, and *subok* requirements are satisfied, the input array is returned instead of a copy.

**Returns**

**arr_t**

[ndarray] Unless `copy` is False and the other conditions for returning the input array are satisfied (see description for `copy` input parameter), *arr_t* is a new array of the same shape as the input array, with dtype, order given by `dtype`, *order*.

**Raises**

**ComplexWarning**

When casting from complex to float or int. To avoid this, one should use `a.real.astype(t)`.

### Notes

Changed in version 1.17.0: Casting between a simple data type and a structured one is possible only for “unsafe” casting. Casting to multiple fields is allowed, but casting from multiple fields is not.

Changed in version 1.9.0: Casting from numeric to string types in ‘safe’ casting mode requires that the string dtype length is long enough to store the max integer/float value converted.

## Examples

```
>>> x = np.array([1, 2, 2.5])
>>> x
array([1. ,  2. ,  2.5])
```

```
>>> x.astype(int)
array([1, 2, 2])
```

method

`matrix.`**`byteswap`**`(`*inplace=False*`)`

Swap the bytes of the array elements

Toggle between low-endian and big-endian data representation by returning a byteswapped array, optionally swapped in-place. Arrays of byte-strings are not swapped. The real and imaginary parts of a complex number are swapped individually.

**Parameters**

**inplace**

[bool, optional] If `True`, swap bytes in-place, default is `False`.

**Returns**

**out**

[ndarray] The byteswapped array. If *inplace* is `True`, this is a view to self.

## Examples

```
>>> A = np.array([1, 256, 8755], dtype=np.int16)
>>> list(map(hex, A))
['0x1', '0x100', '0x2233']
>>> A.byteswap(inplace=True)
array([  256,     1, 13090], dtype=int16)
>>> list(map(hex, A))
['0x100', '0x1', '0x3322']
```

Arrays of byte-strings are not swapped

```
>>> A = np.array([b'ceg', b'fac'])
>>> A.byteswap()
array([b'ceg', b'fac'], dtype='|S3')
```

**`A.newbyteorder().byteswap()` produces an array with the same values**

but different representation in memory

```
>>> A = np.array([1, 2, 3])
>>> A.view(np.uint8)
array([1, 0, 0, 0, 0, 0, 0, 0, 2, 0, 0, 0, 0, 0, 0, 0, 3, 0, 0, 0, 0, 0,
       0, 0], dtype=uint8)
>>> A.newbyteorder().byteswap(inplace=True)
```

(continues on next page)

(continued from previous page)

```
array([1, 2, 3])
>>> A.view(np.uint8)
array([0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 2, 0, 0, 0, 0, 0, 0,
       0, 3], dtype=uint8)
```

method

`matrix.`**`choose`**`(`*choices*, *out=None*, *mode='raise'*`)`

Use an index array to construct a new array from a set of choices.

Refer to `numpy.choose` for full documentation.

**See also:**

***`numpy.choose`***

equivalent function

method

`matrix.`**`clip`**`(`*min=None*, *max=None*, *out=None*, ***kwargs*`)`

Return an array whose values are limited to `[min, max]`. One of max or min must be given.

Refer to `numpy.clip` for full documentation.

**See also:**

***`numpy.clip`***

equivalent function

method

`matrix.`**`compress`**`(`*condition*, *axis=None*, *out=None*`)`

Return selected slices of this array along given axis.

Refer to `numpy.compress` for full documentation.

**See also:**

***`numpy.compress`***

equivalent function

method

`matrix.`**`conj`**`()`

Complex-conjugate all elements.

Refer to `numpy.conjugate` for full documentation.

**See also:**

***`numpy.conjugate`***

equivalent function

method

`matrix.`**`conjugate`**`()`

Return the complex conjugate, element-wise.

Refer to *`numpy.conjugate`* for full documentation.

**See also:**

***`numpy.conjugate`***

equivalent function

method

`matrix.`**`copy`**`(`*order='C'*`)`

Return a copy of the array.

**Parameters**

**order**

[{'C', 'F', 'A', 'K'}, optional] Controls the memory layout of the copy. 'C' means C-order, 'F' means F-order, 'A' means 'F' if *a* is Fortran contiguous, 'C' otherwise. 'K' means match the layout of *a* as closely as possible. (Note that this function and *`numpy.copy`* are very similar, but have different default values for their order= arguments.)

**See also:**

*`numpy.copy`*, *`numpy.copyto`*

## Examples

```
>>> x = np.array([[1,2,3],[4,5,6]], order='F')
```

```
>>> y = x.copy()
```

```
>>> x.fill(0)
```

```
>>> x
array([[0, 0, 0],
       [0, 0, 0]])
```

```
>>> y
array([[1, 2, 3],
       [4, 5, 6]])
```

```
>>> y.flags['C_CONTIGUOUS']
True
```

method

`matrix.`**`cumprod`**`(`*axis=None*, *dtype=None*, *out=None*`)`

Return the cumulative product of the elements along the given axis.

Refer to *`numpy.cumprod`* for full documentation.

**See also:**

**`numpy.cumprod`**

equivalent function

method

`matrix.`**`cumsum`**(*axis=None*, *dtype=None*, *out=None*)

Return the cumulative sum of the elements along the given axis.

Refer to `numpy.cumsum` for full documentation.

**See also:**

**`numpy.cumsum`**

equivalent function

method

`matrix.`**`diagonal`**(*offset=0*, *axis1=0*, *axis2=1*)

Return specified diagonals. In NumPy 1.9 the returned array is a read-only view instead of a copy as in previous NumPy versions. In a future version the read-only restriction will be removed.

Refer to `numpy.diagonal` for full documentation.

**See also:**

**`numpy.diagonal`**

equivalent function

method

`matrix.`**`dot`**(*b*, *out=None*)

Dot product of two arrays.

Refer to `numpy.dot` for full documentation.

**See also:**

**`numpy.dot`**

equivalent function

### Examples

```
>>> a = np.eye(2)
>>> b = np.ones((2, 2)) * 2
>>> a.dot(b)
array([[2.,  2.],
       [2.,  2.]])
```

This array method can be conveniently chained:

```
>>> a.dot(b).dot(b)
array([[8.,  8.],
       [8.,  8.]])
```

method

`matrix.`**`dump`**`(`*file*`)`

Dump a pickle of the array to the specified file. The array can be read back with pickle.load or numpy.load.

**Parameters**

**file**

[str or Path] A string naming the dump file.

Changed in version 1.17.0: `pathlib.Path` objects are now accepted.

method

`matrix.`**`dumps`**`()`

Returns the pickle of the array as a string. pickle.loads or numpy.loads will convert the string back to an array.

**Parameters**

**None**

method

`matrix.`**`fill`**`(`*value*`)`

Fill the array with a scalar value.

**Parameters**

**value**

[scalar] All elements of *a* will be assigned this value.

### Examples

```
>>> a = np.array([1, 2])
>>> a.fill(0)
>>> a
array([0, 0])
>>> a = np.empty(2)
>>> a.fill(1)
>>> a
array([1.,  1.])
```

method

`matrix.`**`flatten`**`(`*self*, *order='C'*`)`

Return a flattened copy of the matrix.

All *N* elements of the matrix are placed into a single row.

**Parameters**

**order**

[{'C', 'F', 'A', 'K'}, optional] 'C' means to flatten in row-major (C-style) order. 'F' means to flatten in column-major (Fortran-style) order. 'A' means to flatten in column-major order if *m* is Fortran *contiguous* in memory, row-major order otherwise. 'K' means to flatten *m* in the order the elements occur in memory. The default is 'C'.

**Returns**

**y**

[matrix] A copy of the matrix, flattened to a *(1, N)* matrix where *N* is the number of elements in the original matrix.

**See also:**

***ravel***

Return a flattened array.

**flat**

A 1-D flat iterator over the matrix.

## Examples

```
>>> m = np.matrix([[1,2], [3,4]])
>>> m.flatten()
matrix([[1, 2, 3, 4]])
>>> m.flatten('F')
matrix([[1, 3, 2, 4]])
```

method

`matrix.`**`getA`**(*self*)

Return *self* as an `ndarray` object.

Equivalent to `np.asarray(self)`.

**Parameters**

**None**

**Returns**

**ret**

[ndarray] *self* as an `ndarray`

## Examples

```
>>> x = np.matrix(np.arange(12).reshape((3,4))); x
matrix([[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [ 8,  9, 10, 11]])
>>> x.getA()
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

method

`matrix.`**`getA1`**(*self*)

Return *self* as a flattened `ndarray`.

Equivalent to `np.asarray(x).ravel()`

**Parameters**

**None**

**Returns**

**ret**

[ndarray] *self*, 1-D, as an `ndarray`

### Examples

```
>>> x = np.matrix(np.arange(12).reshape((3,4))); x
matrix([[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [ 8,  9, 10, 11]])
>>> x.getA1()
array([ 0,  1,  2, ...,  9, 10, 11])
```

method

`matrix.`**`getH`**`(`*self*`)`

Returns the (complex) conjugate transpose of *self*.

Equivalent to `np.transpose(self)` if *self* is real-valued.

**Parameters**

**None**

**Returns**

**ret**

[matrix object] complex conjugate transpose of *self*

### Examples

```
>>> x = np.matrix(np.arange(12).reshape((3,4)))
>>> z = x - 1j*x; z
matrix([[  0. +0.j,   1. -1.j,   2. -2.j,   3. -3.j],
        [  4. -4.j,   5. -5.j,   6. -6.j,   7. -7.j],
        [  8. -8.j,   9. -9.j,  10.-10.j,  11.-11.j]])
>>> z.getH()
matrix([[ 0. -0.j,  4. +4.j,  8. +8.j],
        [ 1. +1.j,  5. +5.j,  9. +9.j],
        [ 2. +2.j,  6. +6.j, 10.+10.j],
        [ 3. +3.j,  7. +7.j, 11.+11.j]])
```

method

`matrix.`**`getI`**`(`*self*`)`

Returns the (multiplicative) inverse of invertible *self*.

**Parameters**

**None**

**Returns**

**ret**

[matrix object] If *self* is non-singular, *ret* is such that `ret * self` == `self * ret` == `np.matrix(np.eye(self[0,:].size))` all return `True`.

**Raises**

**numpy.linalg.LinAlgError: Singular matrix**

If *self* is singular.

**See also:**

*`linalg.inv`*

### Examples

```
>>> m = np.matrix('[1, 2; 3, 4]'); m
matrix([[1, 2],
        [3, 4]])
>>> m.getI()
matrix([[-2. ,  1. ],
        [ 1.5, -0.5]])
>>> m.getI() * m
matrix([[ 1.,  0.], # may vary
        [ 0.,  1.]])
```

method

`matrix.`**`getT`**`(`*self*`)`

Returns the transpose of the matrix.

Does *not* conjugate! For the complex conjugate transpose, use `.H`.

**Parameters**

**None**

**Returns**

**ret**

[matrix object] The (non-conjugated) transpose of the matrix.

**See also:**

*`transpose`*, *`getH`*

## Examples

```
>>> m = np.matrix('[1, 2; 3, 4]')
>>> m
matrix([[1, 2],
        [3, 4]])
>>> m.getT()
matrix([[1, 3],
        [2, 4]])
```

method

`matrix.`**`getfield`**`(`*dtype*, *offset=0*`)`

Returns a field of the given array as a certain type.

A field is a view of the array data with a given data-type. The values in the view are determined by the given type and the offset into the current array in bytes. The offset needs to be such that the view dtype fits in the array dtype; for example an array of dtype complex128 has 16-byte elements. If taking a view with a 32-bit integer (4 bytes), the offset needs to be between 0 and 12 bytes.

**Parameters**

**dtype**

[str or dtype] The data type of the view. The dtype size of the view can not be larger than that of the array itself.

**offset**

[int] Number of bytes to skip before beginning the element view.

## Examples

```
>>> x = np.diag([1.+1.j]*2)
>>> x[1, 1] = 2 + 4.j
>>> x
array([[1.+1.j,  0.+0.j],
       [0.+0.j,  2.+4.j]])
>>> x.getfield(np.float64)
array([[1.,  0.],
       [0.,  2.]])
```

By choosing an offset of 8 bytes we can select the complex part of the array for our view:

```
>>> x.getfield(np.float64, offset=8)
array([[1.,  0.],
       [0.,  4.]])
```

method

`matrix.`**`item`**`(`**args*`)`

Copy an element of an array to a standard Python scalar and return it.

**Parameters**

***args**

[Arguments (variable number and type)]

- none: in this case, the method only works for arrays with one element (*a.size == 1*), which element is copied into a standard Python scalar object and returned.
- int_type: this argument is interpreted as a flat index into the array, specifying which element to copy and return.
- tuple of int_types: functions as does a single int_type argument, except that the argument is interpreted as an nd-index into the array.

**Returns**

**z**

[Standard Python scalar object] A copy of the specified element of the array as a suitable Python scalar

## Notes

When the data type of *a* is longdouble or clongdouble, item() returns a scalar array object because there is no available Python scalar that would not lose information. Void arrays return a buffer object for item(), unless fields are defined, in which case a tuple is returned.

`item` is very similar to a[args], except, instead of an array scalar, a standard Python scalar is returned. This can be useful for speeding up access to elements of the array and doing arithmetic on elements of the array using Python's optimized math.

## Examples

```
>>> np.random.seed(123)
>>> x = np.random.randint(9, size=(3, 3))
>>> x
array([[2, 2, 6],
       [1, 3, 6],
       [1, 0, 1]])
>>> x.item(3)
1
>>> x.item(7)
0
>>> x.item((0, 1))
2
>>> x.item((2, 2))
1
```

method

`matrix.`**`itemset`**`(*args)`

Insert scalar into an array (scalar is cast to array's dtype, if possible)

There must be at least 1 argument, and define the last argument as *item*. Then, `a.itemset(*args)` is equivalent to but faster than `a[args] = item`. The item should be a scalar value and *args* must select a single item in the array *a*.

**Parameters**

***args**

[Arguments] If one argument: a scalar, only used in case *a* is of size 1. If two arguments: the last argument is the value to be set and must be a scalar, the first argument specifies a single array element location. It is either an int or a tuple.

### Notes

Compared to indexing syntax, `itemset` provides some speed increase for placing a scalar into a particular location in an `ndarray`, if you must do this. However, generally this is discouraged: among other problems, it complicates the appearance of the code. Also, when using `itemset` (and `item`) inside a loop, be sure to assign the methods to a local variable to avoid the attribute look-up at each loop iteration.

### Examples

```
>>> np.random.seed(123)
>>> x = np.random.randint(9, size=(3, 3))
>>> x
array([[2, 2, 6],
       [1, 3, 6],
       [1, 0, 1]])
>>> x.itemset(4, 0)
>>> x.itemset((2, 2), 9)
>>> x
array([[2, 2, 6],
       [1, 0, 6],
       [1, 0, 9]])
```

method

`matrix.`**`max`**`(`*self*, *axis=None*, *out=None*`)`

Return the maximum value along an axis.

**Parameters**

**See ‘amax‘ for complete descriptions**

**See also:**

`amax`, `ndarray.max`

### Notes

This is the same as `ndarray.max`, but returns a `matrix` object where `ndarray.max` would return an ndarray.

## Examples

```
>>> x = np.matrix(np.arange(12).reshape((3,4))); x
matrix([[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [ 8,  9, 10, 11]])
>>> x.max()
11
>>> x.max(0)
matrix([[ 8,  9, 10, 11]])
>>> x.max(1)
matrix([[ 3],
        [ 7],
        [11]])
```

method

`matrix.`**`mean`**(*self*, *axis=None*, *dtype=None*, *out=None*)

Returns the average of the matrix elements along the given axis.

Refer to *`numpy.mean`* for full documentation.

**See also:**

*`numpy.mean`*

## Notes

Same as *`ndarray.mean`* except that, where that returns an *`ndarray`*, this returns a *`matrix`* object.

## Examples

```
>>> x = np.matrix(np.arange(12).reshape((3, 4)))
>>> x
matrix([[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [ 8,  9, 10, 11]])
>>> x.mean()
5.5
>>> x.mean(0)
matrix([[4., 5., 6., 7.]])
>>> x.mean(1)
matrix([[ 1.5],
        [ 5.5],
        [ 9.5]])
```

method

`matrix.`**`min`**(*self*, *axis=None*, *out=None*)

Return the minimum value along an axis.

**Parameters**

**See 'amin' for complete descriptions.**

**See also:**

*`amin`*, *`ndarray.min`*

## Notes

This is the same as `ndarray.min`, but returns a `matrix` object where `ndarray.min` would return an ndarray.

## Examples

```
>>> x = -np.matrix(np.arange(12).reshape((3,4))); x
matrix([[  0,  -1,  -2,  -3],
        [ -4,  -5,  -6,  -7],
        [ -8,  -9, -10, -11]])
>>> x.min()
-11
>>> x.min(0)
matrix([[ -8,  -9, -10, -11]])
>>> x.min(1)
matrix([[ -3],
        [ -7],
        [-11]])
```

method

`matrix.`**`newbyteorder`**(*new_order='S'*)

Return the array with the same data viewed with a different byte order.

Equivalent to:

```
arr.view(arr.dtype.newbytorder(new_order))
```

Changes are also made in all fields and sub-arrays of the array data type.

**Parameters**

**new_order**

[string, optional] Byte order to force; a value from the byte order specifications below. *new_order* codes can be any of:

- 'S' - swap dtype from current to opposite endian
- {'<', 'L'} - little endian
- {'>', 'B'} - big endian
- {'=', 'N'} - native order
- {'|', 'I'} - ignore (no change to byte order)

The default value ('S') results in swapping the current byte order. The code does a case-insensitive check on the first letter of *new_order* for the alternatives above. For example, any of 'B' or 'b' or 'biggish' are valid to specify big-endian.

**Returns**

**new_arr**

[array] New array object with the dtype reflecting given change to the byte order.

method

`matrix.`**`nonzero`**`()`

Return the indices of the elements that are non-zero.

Refer to `numpy.nonzero` for full documentation.

**See also:**

**`numpy.nonzero`**

equivalent function

method

`matrix.`**`partition`**`(`*kth*, *axis=-1*, *kind='introselect'*, *order=None*`)`

Rearranges the elements in the array in such a way that the value of the element in kth position is in the position it would be in a sorted array. All elements smaller than the kth element are moved before this element and all equal or greater are moved behind it. The ordering of the elements in the two partitions is undefined.

New in version 1.8.0.

**Parameters**

**kth**

[int or sequence of ints] Element index to partition by. The kth element value will be in its final sorted position and all smaller elements will be moved before it and all equal or greater elements behind it. The order of all elements in the partitions is undefined. If provided with a sequence of kth it will partition all elements indexed by kth of them into their sorted position at once.

**axis**

[int, optional] Axis along which to sort. Default is -1, which means sort along the last axis.

**kind**

[{'introselect'}, optional] Selection algorithm. Default is 'introselect'.

**order**

[str or list of str, optional] When *a* is an array with fields defined, this argument specifies which fields to compare first, second, etc. A single field can be specified as a string, and not all fields need to be specified, but unspecified fields will still be used, in the order in which they come up in the dtype, to break ties.

**See also:**

**`numpy.partition`**

Return a paritioned copy of an array.

**`argpartition`**

Indirect partition.

**`sort`**

Full sort.

### Notes

See `np.partition` for notes on the different algorithms.

### Examples

```
>>> a = np.array([3, 4, 2, 1])
>>> a.partition(3)
>>> a
array([2, 1, 3, 4])
```

```
>>> a.partition((1, 3))
>>> a
array([1, 2, 3, 4])
```

method

`matrix.`**`prod`**(*self*, *axis=None*, *dtype=None*, *out=None*)

Return the product of the array elements over the given axis.

Refer to *`prod`* for full documentation.

**See also:**

*`prod`*, *`ndarray.prod`*

### Notes

Same as *`ndarray.prod`*, except, where that returns an *`ndarray`*, this returns a *`matrix`* object instead.

### Examples

```
>>> x = np.matrix(np.arange(12).reshape((3,4))); x
matrix([[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [ 8,  9, 10, 11]])
>>> x.prod()
0
>>> x.prod(0)
matrix([[  0,  45, 120, 231]])
>>> x.prod(1)
matrix([[   0],
        [ 840],
        [7920]])
```

method

`matrix.`**`ptp`**(*self*, *axis=None*, *out=None*)

Peak-to-peak (maximum - minimum) value along the given axis.

Refer to *`numpy.ptp`* for full documentation.

**See also:**

*`numpy.ptp`*

### Notes

Same as `ndarray.ptp`, except, where that would return an `ndarray` object, this returns a `matrix` object.

### Examples

```
>>> x = np.matrix(np.arange(12).reshape((3,4))); x
matrix([[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [ 8,  9, 10, 11]])
>>> x.ptp()
11
>>> x.ptp(0)
matrix([[8, 8, 8, 8]])
>>> x.ptp(1)
matrix([[3],
        [3],
        [3]])
```

method

`matrix.`**`put`**(*indices*, *values*, *mode='raise'*)

Set `a.flat[n] = values[n]` for all *n* in indices.

Refer to `numpy.put` for full documentation.

**See also:**

**`numpy.put`**

equivalent function

method

`matrix.`**`ravel`**(*self*, *order='C'*)

Return a flattened matrix.

Refer to `numpy.ravel` for more documentation.

**Parameters**

**order**

[{‘C’, ‘F’, ‘A’, ‘K’}, optional] The elements of *m* are read using this index order. ‘C’ means to index the elements in C-like order, with the last axis index changing fastest, back to the first axis index changing slowest. ‘F’ means to index the elements in Fortran-like index order, with the first index changing fastest, and the last index changing slowest. Note that the ‘C’ and ‘F’ options take no account of the memory layout of the underlying array, and only refer to the order of axis indexing. ‘A’ means to read the elements in Fortran-like index order if *m* is Fortran *contiguous* in memory, C-like order otherwise. ‘K’ means to read the elements in the order they occur in memory, except for reversing the data when strides are negative. By default, ‘C’ index order is used.

**Returns**

**ret**

[matrix] Return the matrix flattened to shape *(1, N)* where *N* is the number of elements in the original matrix. A copy is made only if necessary.

**See also:**

***matrix.flatten***

returns a similar output matrix but always a copy

**matrix.flat**

a flat iterator on the array.

***numpy.ravel***

related function which returns an ndarray

method

`matrix.`**`repeat`**(*repeats*, *axis=None*)

Repeat elements of an array.

Refer to *numpy.repeat* for full documentation.

**See also:**

***numpy.repeat***

equivalent function

method

`matrix.`**`reshape`**(*shape*, *order='C'*)

Returns an array containing the same data with a new shape.

Refer to *numpy.reshape* for full documentation.

**See also:**

***numpy.reshape***

equivalent function

## Notes

Unlike the free function *numpy.reshape*, this method on *ndarray* allows the elements of the shape parameter to be passed in as separate arguments. For example, `a.reshape(10, 11)` is equivalent to `a.reshape((10, 11))`.

method

`matrix.`**`resize`**(*new_shape*, *refcheck=True*)

Change shape and size of array in-place.

**Parameters**

**new_shape**

[tuple of ints, or *n* ints] Shape of resized array.

**refcheck**

[bool, optional] If False, reference count will not be checked. Default is True.

**Returns**

**None**

**Raises**

**ValueError**

If *a* does not own its own data or references or views to it exist, and the data memory must be changed. PyPy only: will always raise if the data memory must be changed, since there is no reliable way to determine if references or views to it exist.

**SystemError**

If the *order* keyword argument is specified. This behaviour is a bug in NumPy.

**See also:**

`resize`

Return a new array with the specified shape.

### Notes

This reallocates space for the data area if necessary.

Only contiguous arrays (data elements consecutive in memory) can be resized.

The purpose of the reference count check is to make sure you do not use this array as a buffer for another Python object and then reallocate the memory. However, reference counts can increase in other ways so if you are sure that you have not shared the memory for this array with another Python object, then you may safely set *refcheck* to False.

### Examples

Shrinking an array: array is flattened (in the order that the data are stored in memory), resized, and reshaped:

```
>>> a = np.array([[0, 1], [2, 3]], order='C')
>>> a.resize((2, 1))
>>> a
array([[0],
       [1]])
```

```
>>> a = np.array([[0, 1], [2, 3]], order='F')
>>> a.resize((2, 1))
>>> a
array([[0],
       [2]])
```

Enlarging an array: as above, but missing entries are filled with zeros:

```
>>> b = np.array([[0, 1], [2, 3]])
>>> b.resize(2, 3) # new_shape parameter doesn't have to be a tuple
>>> b
array([[0, 1, 2],
       [3, 0, 0]])
```

Referencing an array prevents resizing…

```
>>> c = a
>>> a.resize((1, 1))
Traceback (most recent call last):
...
ValueError: cannot resize an array that references or is referenced ...
```

Unless *refcheck* is False:

```
>>> a.resize((1, 1), refcheck=False)
>>> a
array([[0]])
>>> c
array([[0]])
```

method

`matrix.`**`round`**`(`*decimals=0*, *out=None*`)`

Return *a* with each element rounded to the given number of decimals.

Refer to *`numpy.around`* for full documentation.

**See also:**

***`numpy.around`***

equivalent function

method

`matrix.`**`searchsorted`**`(`*v*, *side='left'*, *sorter=None*`)`

Find indices where elements of v should be inserted in a to maintain order.

For full documentation, see *`numpy.searchsorted`*

**See also:**

***`numpy.searchsorted`***

equivalent function

method

`matrix.`**`setfield`**`(`*val*, *dtype*, *offset=0*`)`

Put a value into a specified place in a field defined by a data-type.

Place *val* into *a*'s field defined by *`dtype`* and beginning *offset* bytes into the field.

**Parameters**

**val**

[object] Value to be placed in field.

**dtype**

[dtype object] Data-type of the field in which to place *val*.

**offset**

[int, optional] The number of bytes into the field at which to place *val*.

**Returns**

**None**

**See also:**

*getfield*

## Examples

```
>>> x = np.eye(3)
>>> x.getfield(np.float64)
array([[1.,  0.,  0.],
       [0.,  1.,  0.],
       [0.,  0.,  1.]])
>>> x.setfield(3, np.int32)
>>> x.getfield(np.int32)
array([[3, 3, 3],
       [3, 3, 3],
       [3, 3, 3]], dtype=int32)
>>> x
array([[1.0e+000, 1.5e-323, 1.5e-323],
       [1.5e-323, 1.0e+000, 1.5e-323],
       [1.5e-323, 1.5e-323, 1.0e+000]])
>>> x.setfield(np.eye(3), np.int32)
>>> x
array([[1.,  0.,  0.],
       [0.,  1.,  0.],
       [0.,  0.,  1.]])
```

method

`matrix.`**`setflags`**(*write=None*, *align=None*, *uic=None*)

Set array flags WRITEABLE, ALIGNED, (WRITEBACKIFCOPY and UPDATEIFCOPY), respectively.

These Boolean-valued flags affect how numpy interprets the memory area used by *a* (see Notes below). The ALIGNED flag can only be set to True if the data is actually aligned according to the type. The WRITEBACKIFCOPY and (deprecated) UPDATEIFCOPY flags can never be set to True. The flag WRITEABLE can only be set to True if the array owns its own memory, or the ultimate owner of the memory exposes a writeable buffer interface, or is a string. (The exception for string is made so that unpickling can be done without copying memory.)

**Parameters**

**write**

[bool, optional] Describes whether or not *a* can be written to.

**align**

[bool, optional] Describes whether or not *a* is aligned properly for its type.

**uic**

[bool, optional] Describes whether or not *a* is a copy of another "base" array.

### Notes

Array flags provide information about how the memory area used for the array is to be interpreted. There are 7 Boolean flags in use, only four of which can be changed by the user: WRITEBACKIFCOPY, UPDATEIFCOPY, WRITEABLE, and ALIGNED.

WRITEABLE (W) the data area can be written to;

ALIGNED (A) the data and strides are aligned appropriately for the hardware (as determined by the compiler);

UPDATEIFCOPY (U) (deprecated), replaced by WRITEBACKIFCOPY;

WRITEBACKIFCOPY (X) this array is a copy of some other array (referenced by .base). When the C-API function PyArray_ResolveWritebackIfCopy is called, the base array will be updated with the contents of this array.

All flags can be accessed using the single (upper case) letter as well as the full name.

### Examples

```
>>> y = np.array([[3, 1, 7],
...               [2, 0, 0],
...               [8, 5, 9]])
>>> y
array([[3, 1, 7],
       [2, 0, 0],
       [8, 5, 9]])
>>> y.flags
  C_CONTIGUOUS : True
  F_CONTIGUOUS : False
  OWNDATA : True
  WRITEABLE : True
  ALIGNED : True
  WRITEBACKIFCOPY : False
  UPDATEIFCOPY : False
>>> y.setflags(write=0, align=0)
>>> y.flags
  C_CONTIGUOUS : True
  F_CONTIGUOUS : False
  OWNDATA : True
  WRITEABLE : False
  ALIGNED : False
  WRITEBACKIFCOPY : False
  UPDATEIFCOPY : False
>>> y.setflags(uic=1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: cannot set WRITEBACKIFCOPY flag to True
```

method

`matrix.`**`sort`** (*axis=-1*, *kind=None*, *order=None*)

Sort an array in-place. Refer to `numpy.sort` for full documentation.

**Parameters**

**axis**

[int, optional] Axis along which to sort. Default is -1, which means sort along the last axis.

**kind**

[{‘quicksort’, ‘mergesort’, ‘heapsort’, ‘stable’}, optional] Sorting algorithm. The default is ‘quicksort’. Note that both ‘stable’ and ‘mergesort’ use timsort under the covers and, in general, the actual implementation will vary with datatype. The ‘mergesort’ option is retained for backwards compatibility.

Changed in version 1.15.0.: The ‘stable’ option was added.

**order**

[str or list of str, optional] When *a* is an array with fields defined, this argument specifies which fields to compare first, second, etc. A single field can be specified as a string, and not all fields need be specified, but unspecified fields will still be used, in the order in which they come up in the dtype, to break ties.

**See also:**

**`numpy.sort`**

Return a sorted copy of an array.

**`numpy.argsort`**

Indirect sort.

**`numpy.lexsort`**

Indirect stable sort on multiple keys.

**`numpy.searchsorted`**

Find elements in sorted array.

**`numpy.partition`**

Partial sort.

## Notes

See `numpy.sort` for notes on the different sorting algorithms.

## Examples

```
>>> a = np.array([[1,4], [3,1]])
>>> a.sort(axis=1)
>>> a
array([[1, 4],
       [1, 3]])
>>> a.sort(axis=0)
>>> a
array([[1, 3],
       [1, 4]])
```

Use the *order* keyword to specify a field to use when sorting a structured array:

```
>>> a = np.array([('a', 2), ('c', 1)], dtype=[('x', 'S1'), ('y', int)])
>>> a.sort(order='y')
>>> a
array([(b'c', 1), (b'a', 2)],
      dtype=[('x', 'S1'), ('y', '<i8')])
```

method

`matrix.`**`squeeze`**(*self*, *axis=None*)

Return a possibly reshaped matrix.

Refer to *`numpy.squeeze`* for more documentation.

**Parameters**

**axis**

[None or int or tuple of ints, optional] Selects a subset of the single-dimensional entries in the shape. If an axis is selected with shape entry greater than one, an error is raised.

**Returns**

**squeezed**

[matrix] The matrix, but as a (1, N) matrix if it had shape (N, 1).

**See also:**

***`numpy.squeeze`***

related function

## Notes

If *m* has a single column then that column is returned as the single row of a matrix. Otherwise *m* is returned. The returned matrix is always either *m* itself or a view into *m*. Supplying an axis keyword argument will not affect the returned matrix but it may cause an error to be raised.

## Examples

```
>>> c = np.matrix([[1], [2]])
>>> c
matrix([[1],
        [2]])
>>> c.squeeze()
matrix([[1, 2]])
>>> r = c.T
>>> r
matrix([[1, 2]])
>>> r.squeeze()
matrix([[1, 2]])
>>> m = np.matrix([[1, 2], [3, 4]])
>>> m.squeeze()
matrix([[1, 2],
        [3, 4]])
```

method

`matrix.`**`std`**`(`*self*, *axis=None*, *dtype=None*, *out=None*, *ddof=0*`)`

Return the standard deviation of the array elements along the given axis.

Refer to `numpy.std` for full documentation.

**See also:**

`numpy.std`

### Notes

This is the same as `ndarray.std`, except that where an `ndarray` would be returned, a `matrix` object is returned instead.

### Examples

```
>>> x = np.matrix(np.arange(12).reshape((3, 4)))
>>> x
matrix([[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [ 8,  9, 10, 11]])
>>> x.std()
3.4520525295346629 # may vary
>>> x.std(0)
matrix([[ 3.26598632,  3.26598632,  3.26598632,  3.26598632]]) # may vary
>>> x.std(1)
matrix([[ 1.11803399],
        [ 1.11803399],
        [ 1.11803399]])
```

method

`matrix.`**`sum`**`(`*self*, *axis=None*, *dtype=None*, *out=None*`)`

Returns the sum of the matrix elements, along the given axis.

Refer to `numpy.sum` for full documentation.

**See also:**

`numpy.sum`

### Notes

This is the same as `ndarray.sum`, except that where an `ndarray` would be returned, a `matrix` object is returned instead.

## Examples

```
>>> x = np.matrix([[1, 2], [4, 3]])
>>> x.sum()
10
>>> x.sum(axis=1)
matrix([[3],
        [7]])
>>> x.sum(axis=1, dtype='float')
matrix([[3.],
        [7.]])
>>> out = np.zeros((2, 1), dtype='float')
>>> x.sum(axis=1, dtype='float', out=np.asmatrix(out))
matrix([[3.],
        [7.]])
```

method

`matrix.`**`swapaxes`**`(`*axis1*, *axis2*`)`

Return a view of the array with *axis1* and *axis2* interchanged.

Refer to `numpy.swapaxes` for full documentation.

**See also:**

**`numpy.swapaxes`**

equivalent function

method

`matrix.`**`take`**`(`*indices*, *axis=None*, *out=None*, *mode='raise'*`)`

Return an array formed from the elements of *a* at the given indices.

Refer to `numpy.take` for full documentation.

**See also:**

**`numpy.take`**

equivalent function

method

`matrix.`**`tobytes`**`(`*order='C'*`)`

Construct Python bytes containing the raw data bytes in the array.

Constructs Python bytes showing a copy of the raw contents of data memory. The bytes object can be produced in either 'C' or 'Fortran', or 'Any' order (the default is 'C'-order). 'Any' order means C-order unless the F_CONTIGUOUS flag in the array is set, in which case it means 'Fortran' order.

New in version 1.9.0.

**Parameters**

**order**

[{'C', 'F', None}, optional] Order of the data for multidimensional arrays: C, Fortran, or the same as for the original array.

**Returns**

**s**

[bytes] Python bytes exhibiting a copy of *a*'s raw data.

### Examples

```
>>> x = np.array([[0, 1], [2, 3]], dtype='<u2')
>>> x.tobytes()
b'\x00\x00\x01\x00\x02\x00\x03\x00'
>>> x.tobytes('C') == x.tobytes()
True
>>> x.tobytes('F')
b'\x00\x00\x02\x00\x01\x00\x03\x00'
```

method

`matrix.`**`tofile`**(*fid*, *sep=""*, *format="%s"*)

Write array to a file as text or binary (default).

Data is always written in 'C' order, independent of the order of *a*. The data produced by this method can be recovered using the function fromfile().

**Parameters**

**fid**

[file or str or Path] An open file object, or a string containing a filename.

Changed in version 1.17.0: `pathlib.Path` objects are now accepted.

**sep**

[str] Separator between array items for text output. If "" (empty), a binary file is written, equivalent to `file.write(a.tobytes())`.

**format**

[str] Format string for text file output. Each entry in the array is formatted to text by first converting it to the closest Python type, and then using "format" % item.

### Notes

This is a convenience function for quick storage of array data. Information on endianness and precision is lost, so this method is not a good choice for files intended to archive data or transport data between machines with different endianness. Some of these problems can be overcome by outputting the data as text files, at the expense of speed and file size.

When fid is a file object, array contents are directly written to the file, bypassing the file object's `write` method. As a result, tofile cannot be used with files objects supporting compression (e.g., GzipFile) or file-like objects that do not support `fileno()` (e.g., BytesIO).

method

`matrix.`**`tolist`**(*self*)

Return the matrix as a (possibly nested) list.

See *`ndarray.tolist`* for full documentation.

**See also:**

*`ndarray.tolist`*

## Examples

```
>>> x = np.matrix(np.arange(12).reshape((3,4))); x
matrix([[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [ 8,  9, 10, 11]])
>>> x.tolist()
[[0, 1, 2, 3], [4, 5, 6, 7], [8, 9, 10, 11]]
```

method

`matrix.`**`tostring`**(*order='C'*)

A compatibility alias for `tobytes`, with exactly the same behavior.

Despite its name, it returns *bytes* not `str`s.

Deprecated since version 1.19.0.

method

`matrix.`**`trace`**(*offset=0*, *axis1=0*, *axis2=1*, *dtype=None*, *out=None*)

Return the sum along diagonals of the array.

Refer to `numpy.trace` for full documentation.

**See also:**

**`numpy.trace`**

equivalent function

method

`matrix.`**`transpose`**(**axes*)

Returns a view of the array with axes transposed.

For a 1-D array this has no effect, as a transposed vector is simply the same vector. To convert a 1-D array into a 2D column vector, an additional dimension must be added. *np.atleast2d(a).T* achieves this, as does *a[:, np.newaxis]*. For a 2-D array, this is a standard matrix transpose. For an n-D array, if axes are given, their order indicates how the axes are permuted (see Examples). If axes are not provided and `a.shape = (i[0], i[1], ... i[n-2], i[n-1])`, then `a.transpose().shape = (i[n-1], i[n-2], ... i[1], i[0])`.

**Parameters**

**axes**

[None, tuple of ints, or *n* ints]

- None or no argument: reverses the order of the axes.
- tuple of ints: *i* in the *j*-th place in the tuple means *a*'s *i*-th axis becomes *a.transpose()*'s *j*-th axis.
- *n* ints: same as an n-tuple of the same ints (this form is intended simply as a "convenience" alternative to the tuple form)

**Returns**

**out**

[ndarray] View of *a*, with axes suitably permuted.

**See also:**

**`ndarray.T`**

Array property returning the array transposed.

**`ndarray.reshape`**

Give a new shape to an array without changing its data.

## Examples

```
>>> a = np.array([[1, 2], [3, 4]])
>>> a
array([[1, 2],
       [3, 4]])
>>> a.transpose()
array([[1, 3],
       [2, 4]])
>>> a.transpose((1, 0))
array([[1, 3],
       [2, 4]])
>>> a.transpose(1, 0)
array([[1, 3],
       [2, 4]])
```

method

`matrix.`**`var`**(*self*, *axis=None*, *dtype=None*, *out=None*, *ddof=0*)

Returns the variance of the matrix elements, along the given axis.

Refer to `numpy.var` for full documentation.

**See also:**

`numpy.var`

## Notes

This is the same as `ndarray.var`, except that where an `ndarray` would be returned, a `matrix` object is returned instead.

## Examples

```
>>> x = np.matrix(np.arange(12).reshape((3, 4)))
>>> x
matrix([[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [ 8,  9, 10, 11]])
>>> x.var()
11.916666666666666
>>> x.var(0)
matrix([[ 10.66666667,  10.66666667,  10.66666667,  10.66666667]]) # may vary
>>> x.var(1)
matrix([[1.25],
```

(continues on next page)

(continued from previous page)

```
        [1.25],
        [1.25]])
```

method

`matrix.`**`view`**`(`*[dtype][, type]*`)`

New view of array with the same data.

---

**Note:** Passing None for `dtype` is different from omitting the parameter, since the former invokes `dtype(None)` which is an alias for `dtype('float_')`.

---

**Parameters**

**dtype**

[data-type or ndarray sub-class, optional] Data-type descriptor of the returned view, e.g., float32 or int16. Omitting it results in the view having the same data-type as *a*. This argument can also be specified as an ndarray sub-class, which then specifies the type of the returned object (this is equivalent to setting the `type` parameter).

**type**

[Python type, optional] Type of the returned view, e.g., ndarray or matrix. Again, omission of the parameter results in type preservation.

## Notes

`a.view()` is used two different ways:

`a.view(some_dtype)` or `a.view(dtype=some_dtype)` constructs a view of the array's memory with a different data-type. This can cause a reinterpretation of the bytes of memory.

`a.view(ndarray_subclass)` or `a.view(type=ndarray_subclass)` just returns an instance of *ndarray_subclass* that looks at the same array (same shape, dtype, etc.) This does not cause a reinterpretation of the memory.

For `a.view(some_dtype)`, if `some_dtype` has a different number of bytes per entry than the previous dtype (for example, converting a regular array to a structured array), then the behavior of the view cannot be predicted just from the superficial appearance of `a` (shown by `print(a)`). It also depends on exactly how `a` is stored in memory. Therefore if `a` is C-ordered versus fortran-ordered, versus defined as a slice or transpose, etc., the view may give different results.

## Examples

```
>>> x = np.array([(1, 2)], dtype=[('a', np.int8), ('b', np.int8)])
```

Viewing array data using a different type and dtype:

```
>>> y = x.view(dtype=np.int16, type=np.matrix)
>>> y
matrix([[513]], dtype=int16)
>>> print(type(y))
<class 'numpy.matrix'>
```

Creating a view on a structured array so it can be used in calculations

```
>>> x = np.array([(1, 2),(3,4)], dtype=[('a', np.int8), ('b', np.int8)])
>>> xv = x.view(dtype=np.int8).reshape(-1,2)
>>> xv
array([[1, 2],
       [3, 4]], dtype=int8)
>>> xv.mean(0)
array([2.,  3.])
```

Making changes to the view changes the underlying array

```
>>> xv[0,1] = 20
>>> x
array([(1, 20), (3,  4)], dtype=[('a', 'i1'), ('b', 'i1')])
```

Using a view to convert an array to a recarray:

```
>>> z = x.view(np.recarray)
>>> z.a
array([1, 3], dtype=int8)
```

Views share data:

```
>>> x[0] = (9, 10)
>>> z[0]
(9, 10)
```

Views that change the dtype size (bytes per entry) should normally be avoided on arrays defined by slices, transposes, fortran-ordering, etc.:

```
>>> x = np.array([[1,2,3],[4,5,6]], dtype=np.int16)
>>> y = x[:, 0:2]
>>> y
array([[1, 2],
       [4, 5]], dtype=int16)
>>> y.view(dtype=[('width', np.int16), ('length', np.int16)])
Traceback (most recent call last):
    ...
ValueError: To change to a dtype of a different size, the array must be C-
↪contiguous
>>> z = y.copy()
>>> z.view(dtype=[('width', np.int16), ('length', np.int16)])
array([[(1, 2)],
       [(4, 5)]], dtype=[('width', '<i2'), ('length', '<i2')])
```

`numpy.`**`asmatrix`**(*data*, *dtype=None*)

Interpret the input as a matrix.

Unlike *`matrix`*, *`asmatrix`* does not make a copy if the input is already a matrix or an ndarray. Equivalent to `matrix(data, copy=False)`.

**Parameters**

**data**

[array_like] Input data.

**dtype**

[data-type] Data-type of the output matrix.

**Returns**

**mat**

[matrix] *data* interpreted as a matrix.

## Examples

```
>>> x = np.array([[1, 2], [3, 4]])
```

```
>>> m = np.asmatrix(x)
```

```
>>> x[0,0] = 5
```

```
>>> m
matrix([[5, 2],
        [3, 4]])
```

`numpy.`**`bmat`**(*obj*, *ldict=None*, *gdict=None*)

Build a matrix object from a string, nested sequence, or array.

**Parameters**

**obj**

[str or array_like] Input data. If a string, variables in the current scope may be referenced by name.

**ldict**

[dict, optional] A dictionary that replaces local operands in current frame. Ignored if *obj* is not a string or *gdict* is None.

**gdict**

[dict, optional] A dictionary that replaces global operands in current frame. Ignored if *obj* is not a string.

**Returns**

**out**

[matrix] Returns a matrix object, which is a specialized 2-D array.

**See also:**

***`block`***

A generalization of this function for N-d arrays, that returns normal ndarrays.

### Examples

```
>>> A = np.mat('1 1; 1 1')
>>> B = np.mat('2 2; 2 2')
>>> C = np.mat('3 4; 5 6')
>>> D = np.mat('7 8; 9 0')
```

All the following expressions construct the same block matrix:

```
>>> np.bmat([[A, B], [C, D]])
matrix([[1, 1, 2, 2],
        [1, 1, 2, 2],
        [3, 4, 7, 8],
        [5, 6, 9, 0]])
>>> np.bmat(np.r_[np.c_[A, B], np.c_[C, D]])
matrix([[1, 1, 2, 2],
        [1, 1, 2, 2],
        [3, 4, 7, 8],
        [5, 6, 9, 0]])
>>> np.bmat('A,B; C,D')
matrix([[1, 1, 2, 2],
        [1, 1, 2, 2],
        [3, 4, 7, 8],
        [5, 6, 9, 0]])
```

Example 1: Matrix creation from a string

```
>>> a = np.mat('1 2 3; 4 5 3')
>>> print((a*a.T).I)
    [[ 0.29239766 -0.13450292]
     [-0.13450292  0.08187135]]
```

Example 2: Matrix creation from nested sequence

```
>>> np.mat([[1,5,10],[1.0,3,4j]])
matrix([[  1.+0.j,   5.+0.j,  10.+0.j],
        [  1.+0.j,   3.+0.j,   0.+4.j]])
```

Example 3: Matrix creation from an array

```
>>> np.mat(np.random.rand(3,3)).T
matrix([[4.17022005e-01, 3.02332573e-01, 1.86260211e-01],
        [7.20324493e-01, 1.46755891e-01, 3.45560727e-01],
        [1.14374817e-04, 9.23385948e-02, 3.96767474e-01]])
```

## 1.6.3 Memory-mapped file arrays

Memory-mapped files are useful for reading and/or modifying small segments of a large file with regular layout, without reading the entire file into memory. A simple subclass of the ndarray uses a memory-mapped file for the data buffer of the array. For small files, the over-head of reading the entire file into memory is typically not significant, however for large files using memory mapping can save considerable resources.

Memory-mapped-file arrays have one additional method (besides those they inherit from the ndarray): `.flush()` which must be called manually by the user to ensure that any changes to the array actually get written to disk.

| | |
|---|---|
| *memmap* | Create a memory-map to an array stored in a *binary* file on disk. |
| *memmap.flush*(self) | Write any changes in the array to the file on disk. |

**class** numpy.**memmap**

Create a memory-map to an array stored in a *binary* file on disk.

Memory-mapped files are used for accessing small segments of large files on disk, without reading the entire file into memory. NumPy's memmap's are array-like objects. This differs from Python's `mmap` module, which uses file-like objects.

This subclass of ndarray has some unpleasant interactions with some operations, because it doesn't quite fit properly as a subclass. An alternative to using this subclass is to create the `mmap` object yourself, then create an ndarray with ndarray.__new__ directly, passing the object created in its 'buffer=' parameter.

This class may at some point be turned into a factory function which returns a view into an mmap buffer.

Delete the memmap instance to close the memmap file.

**Parameters**

**filename**

[str, file-like object, or pathlib.Path instance] The file name or file object to be used as the array data buffer.

**dtype**

[data-type, optional] The data-type used to interpret the file contents. Default is `uint8`.

**mode**

[{'r+', 'r', 'w+', 'c'}, optional] The file is opened in this mode:

| | |
|---|---|
| 'r' | Open existing file for reading only. |
| 'r+' | Open existing file for reading and writing. |
| 'w+' | Create or overwrite existing file for reading and writing. |
| 'c' | Copy-on-write: assignments affect data in memory, but changes are not saved to disk. The file on disk is read-only. |

Default is 'r+'.

**offset**

[int, optional] In the file, array data starts at this offset. Since *offset* is measured in bytes, it should normally be a multiple of the byte-size of *dtype*. When `mode != 'r'`, even positive offsets beyond end of file are valid; The file will be extended to accommodate the additional data. By default, `memmap` will start at the beginning of the file, even if `filename` is a file pointer `fp` and `fp.tell() != 0`.

**shape**

[tuple, optional] The desired shape of the array. If `mode == 'r'` and the number of remaining bytes after *offset* is not a multiple of the byte-size of *dtype*, you must specify *shape*. By default, the returned array will be 1-D with the number of elements determined by file size and data-type.

**order**

[{'C', 'F'}, optional] Specify the order of the ndarray memory layout: row-major, C-style or column-major, Fortran-style. This only has an effect if the shape is greater than 1-D. The default order is 'C'.

**See also:**

**`lib.format.open_memmap`**

Create or load a memory-mapped `.npy` file.

## Notes

The memmap object can be used anywhere an ndarray is accepted. Given a memmap `fp`, `isinstance(fp, numpy.ndarray)` returns `True`.

Memory-mapped files cannot be larger than 2GB on 32-bit systems.

When a memmap causes a file to be created or extended beyond its current size in the filesystem, the contents of the new part are unspecified. On systems with POSIX filesystem semantics, the extended part will be filled with zero bytes.

## Examples

```
>>> data = np.arange(12, dtype='float32')
>>> data.resize((3,4))
```

This example uses a temporary file so that doctest doesn't write files to your directory. You would use a 'normal' filename.

```
>>> from tempfile import mkdtemp
>>> import os.path as path
>>> filename = path.join(mkdtemp(), 'newfile.dat')
```

Create a memmap with dtype and shape that matches our data:

```
>>> fp = np.memmap(filename, dtype='float32', mode='w+', shape=(3,4))
>>> fp
memmap([[0., 0., 0., 0.],
        [0., 0., 0., 0.],
        [0., 0., 0., 0.]], dtype=float32)
```

Write data to memmap array:

```
>>> fp[:] = data[:]
>>> fp
memmap([[  0.,   1.,   2.,   3.],
        [  4.,   5.,   6.,   7.],
        [  8.,   9.,  10.,  11.]], dtype=float32)
```

```
>>> fp.filename == path.abspath(filename)
True
```

Deletion flushes memory changes to disk before removing the object:

```
>>> del fp
```

Load the memmap and verify data was stored:

```
>>> newfp = np.memmap(filename, dtype='float32', mode='r', shape=(3,4))
>>> newfp
memmap([[  0.,   1.,   2.,   3.],
        [  4.,   5.,   6.,   7.],
        [  8.,   9.,  10.,  11.]], dtype=float32)
```

Read-only memmap:

```
>>> fpr = np.memmap(filename, dtype='float32', mode='r', shape=(3,4))
>>> fpr.flags.writeable
False
```

Copy-on-write memmap:

```
>>> fpc = np.memmap(filename, dtype='float32', mode='c', shape=(3,4))
>>> fpc.flags.writeable
True
```

It's possible to assign to copy-on-write array, but values are only written into the memory copy of the array, and not written to disk:

```
>>> fpc
memmap([[  0.,   1.,   2.,   3.],
        [  4.,   5.,   6.,   7.],
        [  8.,   9.,  10.,  11.]], dtype=float32)
>>> fpc[0,:] = 0
>>> fpc
memmap([[  0.,   0.,   0.,   0.],
        [  4.,   5.,   6.,   7.],
        [  8.,   9.,  10.,  11.]], dtype=float32)
```

File on disk is unchanged:

```
>>> fpr
memmap([[  0.,   1.,   2.,   3.],
        [  4.,   5.,   6.,   7.],
        [  8.,   9.,  10.,  11.]], dtype=float32)
```

Offset into a memmap:

```
>>> fpo = np.memmap(filename, dtype='float32', mode='r', offset=16)
>>> fpo
memmap([  4.,   5.,   6.,   7.,   8.,   9.,  10.,  11.], dtype=float32)
```

**Attributes**

**filename**

[str or pathlib.Path instance] Path to the mapped file.

**offset**

[int] Offset position in the file.

**mode**

[str] File mode.

### Methods

| | |
|---|---|
| *flush*(self) | Write any changes in the array to the file on disk. |

method

`memmap.`**`flush`**`(`*self*`)`

Write any changes in the array to the file on disk.

For further information, see *memmap*.

**Parameters**

**None**

**See also:**

*memmap*

Example:

```
>>> a = np.memmap('newfile.dat', dtype=float, mode='w+', shape=1000)
>>> a[10] = 10.0
>>> a[30] = 30.0
>>> del a
>>> b = np.fromfile('newfile.dat', dtype=float)
>>> print(b[10], b[30])
10.0 30.0
>>> a = np.memmap('newfile.dat', dtype=float)
>>> print(a[10], a[30])
10.0 30.0
```

## 1.6.4 Character arrays (`numpy.char`)

**See also:**

*Creating character arrays (numpy.char)*

**Note:** The *chararray* class exists for backwards compatibility with Numarray, it is not recommended for new development. Starting from numpy 1.4, if one needs arrays of strings, it is recommended to use arrays of *dtype* `object_`, `string_` or `unicode_`, and use the free functions in the *numpy.char* module for fast vectorized string operations.

These are enhanced arrays of either `string_` type or `unicode_` type. These arrays inherit from the *ndarray*, but specially-define the operations `+`, `*`, and `%` on a (broadcasting) element-by-element basis. These operations are not available on the standard *ndarray* of character type. In addition, the *chararray* has all of the standard `string` (and `unicode`) methods, executing them on an element-by-element basis. Perhaps the easiest way to create a chararray is to use *self.view(chararray)* where *self* is an ndarray of str or unicode data-type. However, a chararray can also be created using the *numpy.chararray* constructor, or via the *numpy.char.array* function:

| | |
|---|---|
| `chararray`(shape[, itemsize, unicode, …]) | Provides a convenient view on arrays of string and unicode values. |
| `core.defchararray.array`(obj[, itemsize, …]) | Create a `chararray`. |

**class** `numpy.`**`chararray`**(*shape*, *itemsize=1*, *unicode=False*, *buffer=None*, *offset=0*, *strides=None*, *order=None*)

Provides a convenient view on arrays of string and unicode values.

**Note:** The `chararray` class exists for backwards compatibility with Numarray, it is not recommended for new development. Starting from numpy 1.4, if one needs arrays of strings, it is recommended to use arrays of `dtype` `object_`, `string_` or `unicode_`, and use the free functions in the `numpy.char` module for fast vectorized string operations.

Versus a regular NumPy array of type `str` or `unicode`, this class adds the following functionality:

1) values automatically have whitespace removed from the end when indexed
2) comparison operators automatically remove whitespace from the end when comparing values
3) vectorized string operations are provided as methods (e.g. `endswith`) and infix operators (e.g. `"+", "*", "%"`)

chararrays should be created using `numpy.char.array` or `numpy.char.asarray`, rather than this constructor directly.

This constructor creates the array, using *buffer* (with *offset* and `strides`) if it is not `None`. If *buffer* is `None`, then constructs a new array with `strides` in “C order”, unless both `len(shape) >= 2` and `order='F'`, in which case `strides` is in “Fortran order”.

**Parameters**

**shape**

[tuple] Shape of the array.

**itemsize**

[int, optional] Length of each array element, in number of characters. Default is 1.

**unicode**

[bool, optional] Are the array elements of type unicode (True) or string (False). Default is False.

**buffer**

[object exposing the buffer interface or str, optional] Memory address of the start of the array data. Default is None, in which case a new array is created.

**offset**

[int, optional] Fixed stride displacement from the beginning of an axis? Default is 0. Needs to be >=0.

**strides**

[array_like of ints, optional] Strides for the array (see `ndarray.strides` for full description). Default is None.

**order**

[{'C', 'F'}, optional] The order in which the array data is stored in memory: 'C' -> "row major" order (the default), 'F' -> "column major" (Fortran) order.

## Examples

```
>>> charar = np.chararray((3, 3))
>>> charar[:] = 'a'
>>> charar
chararray([[b'a', b'a', b'a'],
           [b'a', b'a', b'a'],
           [b'a', b'a', b'a']], dtype='|S1')
```

```
>>> charar = np.chararray(charar.shape, itemsize=5)
>>> charar[:] = 'abc'
>>> charar
chararray([[b'abc', b'abc', b'abc'],
           [b'abc', b'abc', b'abc'],
           [b'abc', b'abc', b'abc']], dtype='|S5')
```

**Attributes**

**`T`**

The transposed array.

**`base`**

Base object if memory is from some other object.

**`ctypes`**

An object to simplify the interaction of the array with the ctypes module.

**`data`**

Python buffer object pointing to the start of the array's data.

***`dtype`***

Data-type of the array's elements.

**`flags`**

Information about the memory layout of the array.

**`flat`**

A 1-D iterator over the array.

***`imag`***

The imaginary part of the array.

**`itemsize`**

Length of one array element in bytes.

**`nbytes`**

Total bytes consumed by the elements of the array.

**ndim**

Number of array dimensions.

***real***

The real part of the array.

***shape***

Tuple of array dimensions.

**size**

Number of elements in the array.

**strides**

Tuple of bytes to step in each dimension when traversing an array.

## Methods

| | |
|---|---|
| *astype*(dtype[, order, casting, subok, copy]) | Copy of the array, cast to a specified type. |
| *argsort*([axis, kind, order]) | Returns the indices that would sort this array. |
| *copy*([order]) | Return a copy of the array. |
| *count*(self, sub[, start, end]) | Returns an array with the number of non-overlapping occurrences of substring *sub* in the range [*start*, *end*]. |
| *decode*(self[, encoding, errors]) | Calls *str.decode* element-wise. |
| *dump*(file) | Dump a pickle of the array to the specified file. |
| *dumps*() | Returns the pickle of the array as a string. |
| *encode*(self[, encoding, errors]) | Calls `str.encode` element-wise. |
| *endswith*(self, suffix[, start, end]) | Returns a boolean array which is *True* where the string element in *self* ends with *suffix*, otherwise *False*. |
| *expandtabs*(self[, tabsize]) | Return a copy of each string element where all tab characters are replaced by one or more spaces. |
| *fill*(value) | Fill the array with a scalar value. |
| *find*(self, sub[, start, end]) | For each element, return the lowest index in the string where substring *sub* is found. |
| *flatten*([order]) | Return a copy of the array collapsed into one dimension. |
| *getfield*(dtype[, offset]) | Returns a field of the given array as a certain type. |
| *index*(self, sub[, start, end]) | Like *find*, but raises *ValueError* when the substring is not found. |
| *isalnum*(self) | Returns true for each element if all characters in the string are alphanumeric and there is at least one character, false otherwise. |
| *isalpha*(self) | Returns true for each element if all characters in the string are alphabetic and there is at least one character, false otherwise. |
| *isdecimal*(self) | For each element in *self*, return True if there are only decimal characters in the element. |
| *isdigit*(self) | Returns true for each element if all characters in the string are digits and there is at least one character, false otherwise. |

Continued on next page

Table 41 – continued from previous page

| | |
|---|---|
| `islower`(self) | Returns true for each element if all cased characters in the string are lowercase and there is at least one cased character, false otherwise. |
| `isnumeric`(self) | For each element in *self*, return True if there are only numeric characters in the element. |
| `isspace`(self) | Returns true for each element if there are only whitespace characters in the string and there is at least one character, false otherwise. |
| `istitle`(self) | Returns true for each element if the element is a titlecased string and there is at least one character, false otherwise. |
| `isupper`(self) | Returns true for each element if all cased characters in the string are uppercase and there is at least one character, false otherwise. |
| `item`(*args) | Copy an element of an array to a standard Python scalar and return it. |
| `join`(self, seq) | Return a string which is the concatenation of the strings in the sequence *seq*. |
| `ljust`(self, width[, fillchar]) | Return an array with the elements of *self* left-justified in a string of length *width*. |
| `lower`(self) | Return an array with the elements of *self* converted to lowercase. |
| `lstrip`(self[, chars]) | For each element in *self*, return a copy with the leading characters removed. |
| `nonzero`() | Return the indices of the elements that are non-zero. |
| `put`(indices, values[, mode]) | Set `a.flat[n] = values[n]` for all *n* in indices. |
| `ravel`([order]) | Return a flattened array. |
| `repeat`(repeats[, axis]) | Repeat elements of an array. |
| `replace`(self, old, new[, count]) | For each element in *self*, return a copy of the string with all occurrences of substring *old* replaced by *new*. |
| `reshape`(shape[, order]) | Returns an array containing the same data with a new shape. |
| `resize`(new_shape[, refcheck]) | Change shape and size of array in-place. |
| `rfind`(self, sub[, start, end]) | For each element in *self*, return the highest index in the string where substring *sub* is found, such that *sub* is contained within [*start*, *end*]. |
| `rindex`(self, sub[, start, end]) | Like `rfind`, but raises *ValueError* when the substring *sub* is not found. |
| `rjust`(self, width[, fillchar]) | Return an array with the elements of *self* right-justified in a string of length *width*. |
| `rsplit`(self[, sep, maxsplit]) | For each element in *self*, return a list of the words in the string, using *sep* as the delimiter string. |
| `rstrip`(self[, chars]) | For each element in *self*, return a copy with the trailing characters removed. |
| `searchsorted`(v[, side, sorter]) | Find indices where elements of v should be inserted in a to maintain order. |
| `setfield`(val, dtype[, offset]) | Put a value into a specified place in a field defined by a data-type. |

Continued on next page

Table 41 – continued from previous page

| | |
|---|---|
| `setflags`([write, align, uic]) | Set array flags WRITEABLE, ALIGNED, (WRITEBACKIFCOPY and UPDATEIFCOPY), respectively. |
| `sort`([axis, kind, order]) | Sort an array in-place. |
| `split`(self[, sep, maxsplit]) | For each element in *self*, return a list of the words in the string, using *sep* as the delimiter string. |
| `splitlines`(self[, keepends]) | For each element in *self*, return a list of the lines in the element, breaking at line boundaries. |
| `squeeze`([axis]) | Remove single-dimensional entries from the shape of *a*. |
| `startswith`(self, prefix[, start, end]) | Returns a boolean array which is *True* where the string element in *self* starts with *prefix*, otherwise *False*. |
| `strip`(self[, chars]) | For each element in *self*, return a copy with the leading and trailing characters removed. |
| `swapaxes`(axis1, axis2) | Return a view of the array with *axis1* and *axis2* interchanged. |
| `swapcase`(self) | For each element in *self*, return a copy of the string with uppercase characters converted to lowercase and vice versa. |
| `take`(indices[, axis, out, mode]) | Return an array formed from the elements of *a* at the given indices. |
| `title`(self) | For each element in *self*, return a titlecased version of the string: words start with uppercase characters, all remaining cased characters are lowercase. |
| `tofile`(fid[, sep, format]) | Write array to a file as text or binary (default). |
| `tolist`() | Return the array as an `a.ndim`-levels deep nested list of Python scalars. |
| `tostring`([order]) | A compatibility alias for `tobytes`, with exactly the same behavior. |
| `translate`(self, table[, deletechars]) | For each element in *self*, return a copy of the string where all characters occurring in the optional argument *deletechars* are removed, and the remaining characters have been mapped through the given translation table. |
| `transpose`(*axes) | Returns a view of the array with axes transposed. |
| `upper`(self) | Return an array with the elements of *self* converted to uppercase. |
| `view`([dtype][, type]) | New view of array with the same data. |
| `zfill`(self, width) | Return the numeric string left-filled with zeros in a string of length *width*. |

method

`chararray.`**`astype`**(*dtype*, *order='K'*, *casting='unsafe'*, *subok=True*, *copy=True*)

Copy of the array, cast to a specified type.

**Parameters**

**dtype**

[str or dtype] Typecode or data-type to which the array is cast.

**order**

[{'C', 'F', 'A', 'K'}, optional] Controls the memory layout order of the result. 'C' means C order, 'F' means Fortran order, 'A' means 'F' order if all the arrays are Fortran contiguous, 'C' order otherwise, and 'K' means as close to the order the array elements appear in memory as possible. Default is 'K'.

**casting**

[{'no', 'equiv', 'safe', 'same_kind', 'unsafe'}, optional] Controls what kind of data casting may occur. Defaults to 'unsafe' for backwards compatibility.

- 'no' means the data types should not be cast at all.
- 'equiv' means only byte-order changes are allowed.
- 'safe' means only casts which can preserve values are allowed.
- 'same_kind' means only safe casts or casts within a kind, like float64 to float32, are allowed.
- 'unsafe' means any data conversions may be done.

**subok**

[bool, optional] If True, then sub-classes will be passed-through (default), otherwise the returned array will be forced to be a base-class array.

**copy**

[bool, optional] By default, astype always returns a newly allocated array. If this is set to false, and the `dtype`, *order*, and *subok* requirements are satisfied, the input array is returned instead of a copy.

**Returns**

**arr_t**

[ndarray] Unless `copy` is False and the other conditions for returning the input array are satisfied (see description for `copy` input parameter), *arr_t* is a new array of the same shape as the input array, with dtype, order given by `dtype`, *order*.

**Raises**

**ComplexWarning**

When casting from complex to float or int. To avoid this, one should use `a.real.astype(t)`.

## Notes

Changed in version 1.17.0: Casting between a simple data type and a structured one is possible only for "unsafe" casting. Casting to multiple fields is allowed, but casting from multiple fields is not.

Changed in version 1.9.0: Casting from numeric to string types in 'safe' casting mode requires that the string dtype length is long enough to store the max integer/float value converted.

## Examples

```
>>> x = np.array([1, 2, 2.5])
>>> x
array([1. ,  2. ,  2.5])
```

```
>>> x.astype(int)
array([1, 2, 2])
```

method

`chararray.`**`argsort`**(*axis=-1*, *kind=None*, *order=None*)

Returns the indices that would sort this array.

Refer to *`numpy.argsort`* for full documentation.

**See also:**

***`numpy.argsort`***

equivalent function

method

`chararray.`**`copy`**(*order='C'*)

Return a copy of the array.

**Parameters**

**order**

[{'C', 'F', 'A', 'K'}, optional] Controls the memory layout of the copy. 'C' means C-order, 'F' means F-order, 'A' means 'F' if *a* is Fortran contiguous, 'C' otherwise. 'K' means match the layout of *a* as closely as possible. (Note that this function and *`numpy.copy`* are very similar, but have different default values for their order= arguments.)

**See also:**

*`numpy.copy`*, *`numpy.copyto`*

## Examples

```
>>> x = np.array([[1,2,3],[4,5,6]], order='F')
```

```
>>> y = x.copy()
```

```
>>> x.fill(0)
```

```
>>> x
array([[0, 0, 0],
       [0, 0, 0]])
```

```
>>> y
array([[1, 2, 3],
       [4, 5, 6]])
```

```
>>> y.flags['C_CONTIGUOUS']
True
```

method

`chararray.`**`count`**`(`*self*, *sub*, *start=0*, *end=None*`)`

Returns an array with the number of non-overlapping occurrences of substring *sub* in the range [*start*, *end*].

**See also:**

*`char.count`*

method

`chararray.`**`decode`**`(`*self*, *encoding=None*, *errors=None*`)`

Calls *str.decode* element-wise.

**See also:**

*`char.decode`*

method

`chararray.`**`dump`**`(`*file*`)`

Dump a pickle of the array to the specified file. The array can be read back with pickle.load or numpy.load.

**Parameters**

**file**

[str or Path] A string naming the dump file.

Changed in version 1.17.0: `pathlib.Path` objects are now accepted.

method

`chararray.`**`dumps`**`()`

Returns the pickle of the array as a string. pickle.loads or numpy.loads will convert the string back to an array.

**Parameters**

**None**

method

`chararray.`**`encode`**`(`*self*, *encoding=None*, *errors=None*`)`

Calls `str.encode` element-wise.

**See also:**

*`char.encode`*

method

`chararray.`**`endswith`**`(`*self*, *suffix*, *start=0*, *end=None*`)`

Returns a boolean array which is *True* where the string element in *self* ends with *suffix*, otherwise *False*.

**See also:**

*`char.endswith`*

method

`chararray.`**`expandtabs`**(*self*, *tabsize=8*)

Return a copy of each string element where all tab characters are replaced by one or more spaces.

**See also:**

*`char.expandtabs`*

method

`chararray.`**`fill`**(*value*)

Fill the array with a scalar value.

**Parameters**

**value**

[scalar] All elements of *a* will be assigned this value.

### Examples

```
>>> a = np.array([1, 2])
>>> a.fill(0)
>>> a
array([0, 0])
>>> a = np.empty(2)
>>> a.fill(1)
>>> a
array([1.,  1.])
```

method

`chararray.`**`find`**(*self*, *sub*, *start=0*, *end=None*)

For each element, return the lowest index in the string where substring *sub* is found.

**See also:**

*`char.find`*

method

`chararray.`**`flatten`**(*order='C'*)

Return a copy of the array collapsed into one dimension.

**Parameters**

**order**

[{'C', 'F', 'A', 'K'}, optional] 'C' means to flatten in row-major (C-style) order. 'F' means to flatten in column-major (Fortran- style) order. 'A' means to flatten in column-major order if *a* is Fortran *contiguous* in memory, row-major order otherwise. 'K' means to flatten *a* in the order the elements occur in memory. The default is 'C'.

**Returns**

**y**

[ndarray] A copy of the input array, flattened to one dimension.

**See also:**

*`ravel`*

Return a flattened array.

**`flat`**

A 1-D flat iterator over the array.

### Examples

```
>>> a = np.array([[1,2], [3,4]])
>>> a.flatten()
array([1, 2, 3, 4])
>>> a.flatten('F')
array([1, 3, 2, 4])
```

method

`chararray.`**`getfield`**`(`*dtype*, *offset=0*`)`

Returns a field of the given array as a certain type.

A field is a view of the array data with a given data-type. The values in the view are determined by the given type and the offset into the current array in bytes. The offset needs to be such that the view dtype fits in the array dtype; for example an array of dtype complex128 has 16-byte elements. If taking a view with a 32-bit integer (4 bytes), the offset needs to be between 0 and 12 bytes.

**Parameters**

**dtype**

[str or dtype] The data type of the view. The dtype size of the view can not be larger than that of the array itself.

**offset**

[int] Number of bytes to skip before beginning the element view.

### Examples

```
>>> x = np.diag([1.+1.j]*2)
>>> x[1, 1] = 2 + 4.j
>>> x
array([[1.+1.j,  0.+0.j],
       [0.+0.j,  2.+4.j]])
>>> x.getfield(np.float64)
array([[1.,  0.],
       [0.,  2.]])
```

By choosing an offset of 8 bytes we can select the complex part of the array for our view:

```
>>> x.getfield(np.float64, offset=8)
array([[1.,  0.],
       [0.,  4.]])
```

method

`chararray.`**`index`**`(`*self*, *sub*, *start=0*, *end=None*`)`

Like *`find`*, but raises *ValueError* when the substring is not found.

**See also:**

`char.index`

method

`chararray.`**`isalnum`**`(`*self*`)`

Returns true for each element if all characters in the string are alphanumeric and there is at least one character, false otherwise.

**See also:**

`char.isalnum`

method

`chararray.`**`isalpha`**`(`*self*`)`

Returns true for each element if all characters in the string are alphabetic and there is at least one character, false otherwise.

**See also:**

`char.isalpha`

method

`chararray.`**`isdecimal`**`(`*self*`)`

For each element in *self*, return True if there are only decimal characters in the element.

**See also:**

`char.isdecimal`

method

`chararray.`**`isdigit`**`(`*self*`)`

Returns true for each element if all characters in the string are digits and there is at least one character, false otherwise.

**See also:**

`char.isdigit`

method

`chararray.`**`islower`**`(`*self*`)`

Returns true for each element if all cased characters in the string are lowercase and there is at least one cased character, false otherwise.

**See also:**

`char.islower`

method

`chararray.`**`isnumeric`**`(`*self*`)`

For each element in *self*, return True if there are only numeric characters in the element.

**See also:**

`char.isnumeric`

method

`chararray.`**`isspace`**`(`*self*`)`

Returns true for each element if there are only whitespace characters in the string and there is at least one character, false otherwise.

**See also:**

*`char.isspace`*

method

`chararray.`**`istitle`**`(`*self*`)`

Returns true for each element if the element is a titlecased string and there is at least one character, false otherwise.

**See also:**

*`char.istitle`*

method

`chararray.`**`isupper`**`(`*self*`)`

Returns true for each element if all cased characters in the string are uppercase and there is at least one character, false otherwise.

**See also:**

*`char.isupper`*

method

`chararray.`**`item`**`(`**args*`)`

Copy an element of an array to a standard Python scalar and return it.

**Parameters**

***args**

[Arguments (variable number and type)]

- none: in this case, the method only works for arrays with one element (*a.size* == *1*), which element is copied into a standard Python scalar object and returned.
- int_type: this argument is interpreted as a flat index into the array, specifying which element to copy and return.
- tuple of int_types: functions as does a single int_type argument, except that the argument is interpreted as an nd-index into the array.

**Returns**

**z**

[Standard Python scalar object] A copy of the specified element of the array as a suitable Python scalar

### Notes

When the data type of *a* is longdouble or clongdouble, item() returns a scalar array object because there is no available Python scalar that would not lose information. Void arrays return a buffer object for item(), unless fields are defined, in which case a tuple is returned.

*`item`* is very similar to a[args], except, instead of an array scalar, a standard Python scalar is returned. This can be useful for speeding up access to elements of the array and doing arithmetic on elements of the array using Python's optimized math.

### Examples

```
>>> np.random.seed(123)
>>> x = np.random.randint(9, size=(3, 3))
>>> x
array([[2, 2, 6],
       [1, 3, 6],
       [1, 0, 1]])
>>> x.item(3)
1
>>> x.item(7)
0
>>> x.item((0, 1))
2
>>> x.item((2, 2))
1
```

method

`chararray.`**`join`**`(`*self*, *seq*`)`

Return a string which is the concatenation of the strings in the sequence *seq*.

**See also:**

*`char.join`*

method

`chararray.`**`ljust`**`(`*self*, *width*, *fillchar=' '*`)`

Return an array with the elements of *self* left-justified in a string of length *width*.

**See also:**

*`char.ljust`*

method

`chararray.`**`lower`**`(`*self*`)`

Return an array with the elements of *self* converted to lowercase.

**See also:**

*`char.lower`*

method

`chararray.`**`lstrip`**`(`*self*, *chars=None*`)`

For each element in *self*, return a copy with the leading characters removed.

**See also:**

*`char.lstrip`*

method

`chararray.`**`nonzero`**`()`

Return the indices of the elements that are non-zero.

Refer to *`numpy.nonzero`* for full documentation.

**See also:**

***`numpy.nonzero`***

equivalent function

method

`chararray.`**`put`**`(`*indices*, *values*, *mode='raise'*`)`

Set `a.flat[n] = values[n]` for all *n* in indices.

Refer to `numpy.put` for full documentation.

**See also:**

**`numpy.put`**

equivalent function

method

`chararray.`**`ravel`**`(`[*order*]`)`

Return a flattened array.

Refer to `numpy.ravel` for full documentation.

**See also:**

**`numpy.ravel`**

equivalent function

**`ndarray.flat`**

a flat iterator on the array.

method

`chararray.`**`repeat`**`(`*repeats*, *axis=None*`)`

Repeat elements of an array.

Refer to `numpy.repeat` for full documentation.

**See also:**

**`numpy.repeat`**

equivalent function

method

`chararray.`**`replace`**`(`*self*, *old*, *new*, *count=None*`)`

For each element in *self*, return a copy of the string with all occurrences of substring *old* replaced by *new*.

**See also:**

`char.replace`

method

`chararray.`**`reshape`**`(`*shape*, *order='C'*`)`

Returns an array containing the same data with a new shape.

Refer to `numpy.reshape` for full documentation.

**See also:**

**`numpy.reshape`**

equivalent function

### Notes

Unlike the free function `numpy.reshape`, this method on `ndarray` allows the elements of the shape parameter to be passed in as separate arguments. For example, `a.reshape(10, 11)` is equivalent to `a.reshape((10, 11))`.

method

`chararray.`**`resize`**`(`*new_shape*, *refcheck=True*`)`

Change shape and size of array in-place.

**Parameters**

**new_shape**

[tuple of ints, or *n* ints] Shape of resized array.

**refcheck**

[bool, optional] If False, reference count will not be checked. Default is True.

**Returns**

**None**

**Raises**

**ValueError**

If *a* does not own its own data or references or views to it exist, and the data memory must be changed. PyPy only: will always raise if the data memory must be changed, since there is no reliable way to determine if references or views to it exist.

**SystemError**

If the *order* keyword argument is specified. This behaviour is a bug in NumPy.

**See also:**

**`resize`**

Return a new array with the specified shape.

### Notes

This reallocates space for the data area if necessary.

Only contiguous arrays (data elements consecutive in memory) can be resized.

The purpose of the reference count check is to make sure you do not use this array as a buffer for another Python object and then reallocate the memory. However, reference counts can increase in other ways so if you are sure that you have not shared the memory for this array with another Python object, then you may safely set *refcheck* to False.

### Examples

Shrinking an array: array is flattened (in the order that the data are stored in memory), resized, and reshaped:

```
>>> a = np.array([[0, 1], [2, 3]], order='C')
>>> a.resize((2, 1))
>>> a
array([[0],
       [1]])
```

```
>>> a = np.array([[0, 1], [2, 3]], order='F')
>>> a.resize((2, 1))
>>> a
array([[0],
       [2]])
```

Enlarging an array: as above, but missing entries are filled with zeros:

```
>>> b = np.array([[0, 1], [2, 3]])
>>> b.resize(2, 3) # new_shape parameter doesn't have to be a tuple
>>> b
array([[0, 1, 2],
       [3, 0, 0]])
```

Referencing an array prevents resizing…

```
>>> c = a
>>> a.resize((1, 1))
Traceback (most recent call last):
...
ValueError: cannot resize an array that references or is referenced ...
```

Unless *refcheck* is False:

```
>>> a.resize((1, 1), refcheck=False)
>>> a
array([[0]])
>>> c
array([[0]])
```

method

`chararray.`**`rfind`**(*self*, *sub*, *start=0*, *end=None*)

For each element in *self*, return the highest index in the string where substring *sub* is found, such that *sub* is contained within [*start*, *end*].

**See also:**

*`char.rfind`*

method

`chararray.`**`rindex`**(*self*, *sub*, *start=0*, *end=None*)

Like *`rfind`*, but raises *ValueError* when the substring *sub* is not found.

**See also:**

*`char.rindex`*

method

`chararray.`**`rjust`**(*self*, *width*, *fillchar=' '*)
Return an array with the elements of *self* right-justified in a string of length *width*.

**See also:**

`char.rjust`

method

`chararray.`**`rsplit`**(*self*, *sep=None*, *maxsplit=None*)
For each element in *self*, return a list of the words in the string, using *sep* as the delimiter string.

**See also:**

`char.rsplit`

method

`chararray.`**`rstrip`**(*self*, *chars=None*)
For each element in *self*, return a copy with the trailing characters removed.

**See also:**

`char.rstrip`

method

`chararray.`**`searchsorted`**(*v*, *side='left'*, *sorter=None*)
Find indices where elements of v should be inserted in a to maintain order.

For full documentation, see `numpy.searchsorted`

**See also:**

**`numpy.searchsorted`**
equivalent function

method

`chararray.`**`setfield`**(*val*, *dtype*, *offset=0*)
Put a value into a specified place in a field defined by a data-type.

Place *val* into *a*'s field defined by `dtype` and beginning *offset* bytes into the field.

**Parameters**

**val**
[object] Value to be placed in field.

**dtype**
[dtype object] Data-type of the field in which to place *val*.

**offset**
[int, optional] The number of bytes into the field at which to place *val*.

**Returns**

**None**

**See also:**

`getfield`

## Examples

```
>>> x = np.eye(3)
>>> x.getfield(np.float64)
array([[1.,  0.,  0.],
       [0.,  1.,  0.],
       [0.,  0.,  1.]])
>>> x.setfield(3, np.int32)
>>> x.getfield(np.int32)
array([[3, 3, 3],
       [3, 3, 3],
       [3, 3, 3]], dtype=int32)
>>> x
array([[1.0e+000, 1.5e-323, 1.5e-323],
       [1.5e-323, 1.0e+000, 1.5e-323],
       [1.5e-323, 1.5e-323, 1.0e+000]])
>>> x.setfield(np.eye(3), np.int32)
>>> x
array([[1.,  0.,  0.],
       [0.,  1.,  0.],
       [0.,  0.,  1.]])
```

method

`chararray.`**`setflags`**(*write=None*, *align=None*, *uic=None*)

Set array flags WRITEABLE, ALIGNED, (WRITEBACKIFCOPY and UPDATEIFCOPY), respectively.

These Boolean-valued flags affect how numpy interprets the memory area used by *a* (see Notes below). The ALIGNED flag can only be set to True if the data is actually aligned according to the type. The WRITEBACKIFCOPY and (deprecated) UPDATEIFCOPY flags can never be set to True. The flag WRITEABLE can only be set to True if the array owns its own memory, or the ultimate owner of the memory exposes a writeable buffer interface, or is a string. (The exception for string is made so that unpickling can be done without copying memory.)

**Parameters**

**write**

[bool, optional] Describes whether or not *a* can be written to.

**align**

[bool, optional] Describes whether or not *a* is aligned properly for its type.

**uic**

[bool, optional] Describes whether or not *a* is a copy of another "base" array.

## Notes

Array flags provide information about how the memory area used for the array is to be interpreted. There are 7 Boolean flags in use, only four of which can be changed by the user: WRITEBACKIFCOPY, UPDATEIFCOPY, WRITEABLE, and ALIGNED.

WRITEABLE (W) the data area can be written to;

ALIGNED (A) the data and strides are aligned appropriately for the hardware (as determined by the compiler);

UPDATEIFCOPY (U) (deprecated), replaced by WRITEBACKIFCOPY;

WRITEBACKIFCOPY (X) this array is a copy of some other array (referenced by .base). When the C-API function PyArray_ResolveWritebackIfCopy is called, the base array will be updated with the contents of this array.

All flags can be accessed using the single (upper case) letter as well as the full name.

### Examples

```
>>> y = np.array([[3, 1, 7],
...               [2, 0, 0],
...               [8, 5, 9]])
>>> y
array([[3, 1, 7],
       [2, 0, 0],
       [8, 5, 9]])
>>> y.flags
  C_CONTIGUOUS : True
  F_CONTIGUOUS : False
  OWNDATA : True
  WRITEABLE : True
  ALIGNED : True
  WRITEBACKIFCOPY : False
  UPDATEIFCOPY : False
>>> y.setflags(write=0, align=0)
>>> y.flags
  C_CONTIGUOUS : True
  F_CONTIGUOUS : False
  OWNDATA : True
  WRITEABLE : False
  ALIGNED : False
  WRITEBACKIFCOPY : False
  UPDATEIFCOPY : False
>>> y.setflags(uic=1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: cannot set WRITEBACKIFCOPY flag to True
```

method

`chararray.`**`sort`**(*axis=-1*, *kind=None*, *order=None*)

Sort an array in-place. Refer to `numpy.sort` for full documentation.

**Parameters**

**axis**

[int, optional] Axis along which to sort. Default is -1, which means sort along the last axis.

**kind**

[{'quicksort', 'mergesort', 'heapsort', 'stable'}, optional] Sorting algorithm. The default is 'quicksort'. Note that both 'stable' and 'mergesort' use timsort under the covers and, in general, the actual implementation will vary with datatype. The 'mergesort' option is retained for backwards compatibility.

Changed in version 1.15.0.: The 'stable' option was added.

**order**

[str or list of str, optional] When *a* is an array with fields defined, this argument specifies which fields to compare first, second, etc. A single field can be specified as a string, and not all fields need be specified, but unspecified fields will still be used, in the order in which they come up in the dtype, to break ties.

**See also:**

**`numpy.sort`**

Return a sorted copy of an array.

**`numpy.argsort`**

Indirect sort.

**`numpy.lexsort`**

Indirect stable sort on multiple keys.

**`numpy.searchsorted`**

Find elements in sorted array.

**`numpy.partition`**

Partial sort.

### Notes

See `numpy.sort` for notes on the different sorting algorithms.

### Examples

```
>>> a = np.array([[1,4], [3,1]])
>>> a.sort(axis=1)
>>> a
array([[1, 4],
       [1, 3]])
>>> a.sort(axis=0)
>>> a
array([[1, 3],
       [1, 4]])
```

Use the *order* keyword to specify a field to use when sorting a structured array:

```
>>> a = np.array([('a', 2), ('c', 1)], dtype=[('x', 'S1'), ('y', int)])
>>> a.sort(order='y')
>>> a
array([(b'c', 1), (b'a', 2)],
      dtype=[('x', 'S1'), ('y', '<i8')])
```

method

`chararray.`**`split`**(*self*, *sep=None*, *maxsplit=None*)

For each element in *self*, return a list of the words in the string, using *sep* as the delimiter string.

**See also:**

`char.split`

method

`chararray.`**`splitlines`**`(`*self*, *keepends=None*`)`

For each element in *self*, return a list of the lines in the element, breaking at line boundaries.

**See also:**

*`char.splitlines`*

method

`chararray.`**`squeeze`**`(`*axis=None*`)`

Remove single-dimensional entries from the shape of *a*.

Refer to *`numpy.squeeze`* for full documentation.

**See also:**

***`numpy.squeeze`***

equivalent function

method

`chararray.`**`startswith`**`(`*self*, *prefix*, *start=0*, *end=None*`)`

Returns a boolean array which is *True* where the string element in *self* starts with *prefix*, otherwise *False*.

**See also:**

*`char.startswith`*

method

`chararray.`**`strip`**`(`*self*, *chars=None*`)`

For each element in *self*, return a copy with the leading and trailing characters removed.

**See also:**

*`char.strip`*

method

`chararray.`**`swapaxes`**`(`*axis1*, *axis2*`)`

Return a view of the array with *axis1* and *axis2* interchanged.

Refer to *`numpy.swapaxes`* for full documentation.

**See also:**

***`numpy.swapaxes`***

equivalent function

method

`chararray.`**`swapcase`**`(`*self*`)`

For each element in *self*, return a copy of the string with uppercase characters converted to lowercase and vice versa.

**See also:**

*`char.swapcase`*

method

`chararray.`**`take`**`(`*indices*, *axis=None*, *out=None*, *mode='raise'*`)`

Return an array formed from the elements of *a* at the given indices.

Refer to `numpy.take` for full documentation.

**See also:**

**`numpy.take`**

equivalent function

method

`chararray.`**`title`**`(`*self*`)`

For each element in *self*, return a titlecased version of the string: words start with uppercase characters, all remaining cased characters are lowercase.

**See also:**

`char.title`

method

`chararray.`**`tofile`**`(`*fid*, *sep=""*, *format="%s"*`)`

Write array to a file as text or binary (default).

Data is always written in 'C' order, independent of the order of *a*. The data produced by this method can be recovered using the function fromfile().

**Parameters**

**fid**

[file or str or Path] An open file object, or a string containing a filename.

Changed in version 1.17.0: `pathlib.Path` objects are now accepted.

**sep**

[str] Separator between array items for text output. If "" (empty), a binary file is written, equivalent to `file.write(a.tobytes())`.

**format**

[str] Format string for text file output. Each entry in the array is formatted to text by first converting it to the closest Python type, and then using "format" % item.

### Notes

This is a convenience function for quick storage of array data. Information on endianness and precision is lost, so this method is not a good choice for files intended to archive data or transport data between machines with different endianness. Some of these problems can be overcome by outputting the data as text files, at the expense of speed and file size.

When fid is a file object, array contents are directly written to the file, bypassing the file object's `write` method. As a result, tofile cannot be used with files objects supporting compression (e.g., GzipFile) or file-like objects that do not support `fileno()` (e.g., BytesIO).

method

`chararray.`**`tolist`**`()`

Return the array as an `a.ndim`-levels deep nested list of Python scalars.

Return a copy of the array data as a (nested) Python list. Data items are converted to the nearest compatible builtin Python type, via the *`item`* function.

If `a.ndim` is 0, then since the depth of the nested list is 0, it will not be a list at all, but a simple Python scalar.

**Parameters**

**none**

**Returns**

**y**

[object, or list of object, or list of list of object, or …] The possibly nested list of array elements.

## Notes

The array may be recreated via `a = np.array(a.tolist())`, although this may sometimes lose precision.

## Examples

For a 1D array, `a.tolist()` is almost the same as `list(a)`, except that `tolist` changes numpy scalars to Python scalars:

```
>>> a = np.uint32([1, 2])
>>> a_list = list(a)
>>> a_list
[1, 2]
>>> type(a_list[0])
<class 'numpy.uint32'>
>>> a_tolist = a.tolist()
>>> a_tolist
[1, 2]
>>> type(a_tolist[0])
<class 'int'>
```

Additionally, for a 2D array, `tolist` applies recursively:

```
>>> a = np.array([[1, 2], [3, 4]])
>>> list(a)
[array([1, 2]), array([3, 4])]
>>> a.tolist()
[[1, 2], [3, 4]]
```

The base case for this recursion is a 0D array:

```
>>> a = np.array(1)
>>> list(a)
Traceback (most recent call last):
  ...
```

(continues on next page)

(continued from previous page)

```
TypeError: iteration over a 0-d array
>>> a.tolist()
1
```

method

`chararray.`**`tostring`**`(`*order='C'*`)`

A compatibility alias for `tobytes`, with exactly the same behavior.

Despite its name, it returns *bytes* not `str`s.

Deprecated since version 1.19.0.

method

`chararray.`**`translate`**`(`*self*, *table*, *deletechars=None*`)`

For each element in *self*, return a copy of the string where all characters occurring in the optional argument *deletechars* are removed, and the remaining characters have been mapped through the given translation table.

**See also:**

*`char.translate`*

method

`chararray.`**`transpose`**`(`**axes*`)`

Returns a view of the array with axes transposed.

For a 1-D array this has no effect, as a transposed vector is simply the same vector. To convert a 1-D array into a 2D column vector, an additional dimension must be added. *np.atleast2d(a).T* achieves this, as does *a[:, np.newaxis]*. For a 2-D array, this is a standard matrix transpose. For an n-D array, if axes are given, their order indicates how the axes are permuted (see Examples). If axes are not provided and `a.shape = (i[0], i[1], ... i[n-2], i[n-1])`, then `a.transpose().shape = (i[n-1], i[n-2], ... i[1], i[0])`.

**Parameters**

**axes**

[None, tuple of ints, or *n* ints]

- None or no argument: reverses the order of the axes.
- tuple of ints: *i* in the *j*-th place in the tuple means *a*'s *i*-th axis becomes *a.transpose()*'s *j*-th axis.
- *n* ints: same as an n-tuple of the same ints (this form is intended simply as a "convenience" alternative to the tuple form)

**Returns**

**out**

[ndarray] View of *a*, with axes suitably permuted.

**See also:**

***`ndarray.T`***

Array property returning the array transposed.

*ndarray.reshape*

Give a new shape to an array without changing its data.

### Examples

```
>>> a = np.array([[1, 2], [3, 4]])
>>> a
array([[1, 2],
       [3, 4]])
>>> a.transpose()
array([[1, 3],
       [2, 4]])
>>> a.transpose((1, 0))
array([[1, 3],
       [2, 4]])
>>> a.transpose(1, 0)
array([[1, 3],
       [2, 4]])
```

method

`chararray.`**`upper`**`(`*self*`)`

Return an array with the elements of *self* converted to uppercase.

**See also:**

*char.upper*

method

`chararray.`**`view`**`(`*[dtype][, type]*`)`

New view of array with the same data.

**Note:** Passing None for `dtype` is different from omitting the parameter, since the former invokes `dtype(None)` which is an alias for `dtype('float_')`.

**Parameters**

**dtype**

[data-type or ndarray sub-class, optional] Data-type descriptor of the returned view, e.g., float32 or int16. Omitting it results in the view having the same data-type as *a*. This argument can also be specified as an ndarray sub-class, which then specifies the type of the returned object (this is equivalent to setting the `type` parameter).

**type**

[Python type, optional] Type of the returned view, e.g., ndarray or matrix. Again, omission of the parameter results in type preservation.

## Notes

`a.view()` is used two different ways:

`a.view(some_dtype)` or `a.view(dtype=some_dtype)` constructs a view of the array's memory with a different data-type. This can cause a reinterpretation of the bytes of memory.

`a.view(ndarray_subclass)` or `a.view(type=ndarray_subclass)` just returns an instance of *ndarray_subclass* that looks at the same array (same shape, dtype, etc.) This does not cause a reinterpretation of the memory.

For `a.view(some_dtype)`, if `some_dtype` has a different number of bytes per entry than the previous dtype (for example, converting a regular array to a structured array), then the behavior of the view cannot be predicted just from the superficial appearance of `a` (shown by `print(a)`). It also depends on exactly how `a` is stored in memory. Therefore if `a` is C-ordered versus fortran-ordered, versus defined as a slice or transpose, etc., the view may give different results.

## Examples

```
>>> x = np.array([(1, 2)], dtype=[('a', np.int8), ('b', np.int8)])
```

Viewing array data using a different type and dtype:

```
>>> y = x.view(dtype=np.int16, type=np.matrix)
>>> y
matrix([[513]], dtype=int16)
>>> print(type(y))
<class 'numpy.matrix'>
```

Creating a view on a structured array so it can be used in calculations

```
>>> x = np.array([(1, 2),(3,4)], dtype=[('a', np.int8), ('b', np.int8)])
>>> xv = x.view(dtype=np.int8).reshape(-1,2)
>>> xv
array([[1, 2],
       [3, 4]], dtype=int8)
>>> xv.mean(0)
array([2.,  3.])
```

Making changes to the view changes the underlying array

```
>>> xv[0,1] = 20
>>> x
array([(1, 20), (3,  4)], dtype=[('a', 'i1'), ('b', 'i1')])
```

Using a view to convert an array to a recarray:

```
>>> z = x.view(np.recarray)
>>> z.a
array([1, 3], dtype=int8)
```

Views share data:

```
>>> x[0] = (9, 10)
>>> z[0]
(9, 10)
```

Views that change the dtype size (bytes per entry) should normally be avoided on arrays defined by slices, transposes, fortran-ordering, etc.:

```
>>> x = np.array([[1,2,3],[4,5,6]], dtype=np.int16)
>>> y = x[:, 0:2]
>>> y
array([[1, 2],
       [4, 5]], dtype=int16)
>>> y.view(dtype=[('width', np.int16), ('length', np.int16)])
Traceback (most recent call last):
    ...
ValueError: To change to a dtype of a different size, the array must be C-
↪contiguous
>>> z = y.copy()
>>> z.view(dtype=[('width', np.int16), ('length', np.int16)])
array([[(1, 2)],
       [(4, 5)]], dtype=[('width', '<i2'), ('length', '<i2')])
```

method

`chararray.`**`zfill`**`(`*self*, *width*`)`

Return the numeric string left-filled with zeros in a string of length *width*.

**See also:**

*char.zfill*

`numpy.core.defchararray.`**`array`**`(`*obj*, *itemsize=None*, *copy=True*, *unicode=None*, *order=None*`)`

Create a `chararray`.

---

**Note:** This class is provided for numarray backward-compatibility. New code (not concerned with numarray compatibility) should use arrays of type `string_` or `unicode_` and use the free functions in `numpy.char` for fast vectorized string operations instead.

---

Versus a regular NumPy array of type *str* or *unicode*, this class adds the following functionality:

1) values automatically have whitespace removed from the end when indexed
2) comparison operators automatically remove whitespace from the end when comparing values
3) vectorized string operations are provided as methods (e.g. *str.endswith*) and infix operators (e.g. `+, *, %`)

**Parameters**

**obj**

[array of str or unicode-like]

**itemsize**

[int, optional] *itemsize* is the number of characters per scalar in the resulting array. If *itemsize* is None, and *obj* is an object array or a Python list, the *itemsize* will be automatically determined. If *itemsize* is provided and *obj* is of type str or unicode, then the *obj* string will be chunked into *itemsize* pieces.

**copy**

[bool, optional] If true (default), then the object is copied. Otherwise, a copy will only be made if __array__ returns a copy, if obj is a nested sequence, or if a copy is needed to satisfy any of the other requirements (*itemsize*, unicode, *order*, etc.).

**unicode**

[bool, optional] When true, the resulting `chararray` can contain Unicode characters, when false only 8-bit characters. If unicode is None and *obj* is one of the following:

- a `chararray`,
- an ndarray of type *str* or *unicode*
- a Python str or unicode object,

then the unicode setting of the output array will be automatically determined.

**order**

[{'C', 'F', 'A'}, optional] Specify the order of the array. If order is 'C' (default), then the array will be in C-contiguous order (last-index varies the fastest). If order is 'F', then the returned array will be in Fortran-contiguous order (first-index varies the fastest). If order is 'A', then the returned array may be in any order (either C-, Fortran-contiguous, or even discontiguous).

Another difference with the standard ndarray of str data-type is that the chararray inherits the feature introduced by Numarray that white-space at the end of any element in the array will be ignored on item retrieval and comparison operations.

### 1.6.5 Record arrays (`numpy.rec`)

**See also:**

*Creating record arrays (numpy.rec)*, *Data type routines*, *Data type objects (dtype)*.

NumPy provides the `recarray` class which allows accessing the fields of a structured array as attributes, and a corresponding scalar data type object `record`.

| | |
|---|---|
| `recarray` | Construct an ndarray that allows field access using attributes. |
| `record` | A data-type scalar that allows field access as attribute lookup. |

**class** `numpy.`**`recarray`**

Construct an ndarray that allows field access using attributes.

Arrays may have a data-types containing fields, analogous to columns in a spread sheet. An example is `[(x, int), (y, float)]`, where each entry in the array is a pair of `(int, float)`. Normally, these attributes are accessed using dictionary lookups such as `arr['x']` and `arr['y']`. Record arrays allow the fields to be accessed as members of the array, using `arr.x` and `arr.y`.

**Parameters**

**shape**

[tuple] Shape of output array.

**dtype**

[data-type, optional] The desired data-type. By default, the data-type is determined from *formats*, *names*, *titles*, *aligned* and *byteorder*.

**formats**

[list of data-types, optional] A list containing the data-types for the different columns, e.g.

`['i4', 'f8', 'i4']`. *formats* does *not* support the new convention of using types directly, i.e. `(int, float, int)`. Note that *formats* must be a list, not a tuple. Given that *formats* is somewhat limited, we recommend specifying *`dtype`* instead.

**names**

[tuple of str, optional] The name of each column, e.g. `('x', 'y', 'z')`.

**buf**

[buffer, optional] By default, a new array is created of the given shape and data-type. If *buf* is specified and is an object exposing the buffer interface, the array will use the memory from the existing buffer. In this case, the *offset* and `strides` keywords are available.

**Returns**

**rec**

[recarray] Empty array of the given shape and type.

**Other Parameters**

**titles**

[tuple of str, optional] Aliases for column names. For example, if *names* were `('x', 'y', 'z')` and *titles* is `('x_coordinate', 'y_coordinate', 'z_coordinate')`, then `arr['x']` is equivalent to both `arr.x` and `arr.x_coordinate`.

**byteorder**

[{'<', '>', '='}, optional] Byte-order for all fields.

**aligned**

[bool, optional] Align the fields in memory as the C-compiler would.

**strides**

[tuple of ints, optional] Buffer (*buf*) is interpreted according to these strides (strides define how many bytes each array element, row, column, etc. occupy in memory).

**offset**

[int, optional] Start reading buffer (*buf*) from this offset onwards.

**order**

[{'C', 'F'}, optional] Row-major (C-style) or column-major (Fortran-style) order.

**See also:**

**`rec.fromrecords`**

Construct a record array from data.

***`record`***

fundamental data-type for *`recarray`*.

***`format_parser`***

determine a data-type from formats, names, titles.

## Notes

This constructor can be compared to `empty`: it creates a new record array but does not fill it with data. To create a record array from data, use one of the following methods:

1. Create a standard ndarray and convert it to a record array, using `arr.view(np.recarray)`
2. Use the *buf* keyword.
3. Use *np.rec.fromrecords*.

## Examples

Create an array with two fields, `x` and `y`:

```
>>> x = np.array([(1.0, 2), (3.0, 4)], dtype=[('x', '<f8'), ('y', '<i8')])
>>> x
array([(1., 2), (3., 4)], dtype=[('x', '<f8'), ('y', '<i8')])
```

```
>>> x['x']
array([1., 3.])
```

View the array as a record array:

```
>>> x = x.view(np.recarray)
```

```
>>> x.x
array([1., 3.])
```

```
>>> x.y
array([2, 4])
```

Create a new, empty record array:

```
>>> np.recarray((2,),
... dtype=[('x', int), ('y', float), ('z', int)])
rec.array([(-1073741821, 1.2249118382103472e-301, 24547520),
       (3471280, 1.2134086255804012e-316, 0)],
      dtype=[('x', '<i4'), ('y', '<f8'), ('z', '<i4')])
```

**Attributes**

**`T`**

The transposed array.

**`base`**

Base object if memory is from some other object.

**`ctypes`**

An object to simplify the interaction of the array with the ctypes module.

**`data`**

Python buffer object pointing to the start of the array's data.

**`dtype`**

Data-type of the array's elements.

**`flags`**

Information about the memory layout of the array.

**`flat`**

A 1-D iterator over the array.

**`imag`**

The imaginary part of the array.

**`itemsize`**

Length of one array element in bytes.

**`nbytes`**

Total bytes consumed by the elements of the array.

**`ndim`**

Number of array dimensions.

**`real`**

The real part of the array.

**`shape`**

Tuple of array dimensions.

**`size`**

Number of elements in the array.

**`strides`**

Tuple of bytes to step in each dimension when traversing an array.

## Methods

| | |
|---|---|
| `all`([axis, out, keepdims]) | Returns True if all elements evaluate to True. |
| `any`([axis, out, keepdims]) | Returns True if any of the elements of *a* evaluate to True. |
| `argmax`([axis, out]) | Return indices of the maximum values along the given axis. |
| `argmin`([axis, out]) | Return indices of the minimum values along the given axis of *a*. |
| `argpartition`(kth[, axis, kind, order]) | Returns the indices that would partition this array. |
| `argsort`([axis, kind, order]) | Returns the indices that would sort this array. |
| `astype`(dtype[, order, casting, subok, copy]) | Copy of the array, cast to a specified type. |
| `byteswap`([inplace]) | Swap the bytes of the array elements |
| `choose`(choices[, out, mode]) | Use an index array to construct a new array from a set of choices. |
| `clip`([min, max, out]) | Return an array whose values are limited to `[min, max]`. |

Continued on next page

Table 43 – continued from previous page

| | |
|---|---|
| `compress`(condition[, axis, out]) | Return selected slices of this array along given axis. |
| `conj`() | Complex-conjugate all elements. |
| `conjugate`() | Return the complex conjugate, element-wise. |
| `copy`([order]) | Return a copy of the array. |
| `cumprod`([axis, dtype, out]) | Return the cumulative product of the elements along the given axis. |
| `cumsum`([axis, dtype, out]) | Return the cumulative sum of the elements along the given axis. |
| `diagonal`([offset, axis1, axis2]) | Return specified diagonals. |
| `dot`(b[, out]) | Dot product of two arrays. |
| `dump`(file) | Dump a pickle of the array to the specified file. |
| `dumps`() | Returns the pickle of the array as a string. |
| `fill`(value) | Fill the array with a scalar value. |
| `flatten`([order]) | Return a copy of the array collapsed into one dimension. |
| `getfield`(dtype[, offset]) | Returns a field of the given array as a certain type. |
| `item`(*args) | Copy an element of an array to a standard Python scalar and return it. |
| `itemset`(*args) | Insert scalar into an array (scalar is cast to array's dtype, if possible) |
| `max`([axis, out, keepdims, initial, where]) | Return the maximum along a given axis. |
| `mean`([axis, dtype, out, keepdims]) | Returns the average of the array elements along given axis. |
| `min`([axis, out, keepdims, initial, where]) | Return the minimum along a given axis. |
| `newbyteorder`([new_order]) | Return the array with the same data viewed with a different byte order. |
| `nonzero`() | Return the indices of the elements that are non-zero. |
| `partition`(kth[, axis, kind, order]) | Rearranges the elements in the array in such a way that the value of the element in kth position is in the position it would be in a sorted array. |
| `prod`([axis, dtype, out, keepdims, initial, …]) | Return the product of the array elements over the given axis |
| `ptp`([axis, out, keepdims]) | Peak to peak (maximum - minimum) value along a given axis. |
| `put`(indices, values[, mode]) | Set `a.flat[n] = values[n]` for all *n* in indices. |
| `ravel`([order]) | Return a flattened array. |
| `repeat`(repeats[, axis]) | Repeat elements of an array. |
| `reshape`(shape[, order]) | Returns an array containing the same data with a new shape. |
| `resize`(new_shape[, refcheck]) | Change shape and size of array in-place. |
| `round`([decimals, out]) | Return *a* with each element rounded to the given number of decimals. |
| `searchsorted`(v[, side, sorter]) | Find indices where elements of v should be inserted in a to maintain order. |
| `setfield`(val, dtype[, offset]) | Put a value into a specified place in a field defined by a data-type. |
| `setflags`([write, align, uic]) | Set array flags WRITEABLE, ALIGNED, (WRITEBACKIFCOPY and UPDATEIFCOPY), respectively. |
| `sort`([axis, kind, order]) | Sort an array in-place. |

Continued on next page

Table 43 – continued from previous page

| | |
|---|---|
| `squeeze`([axis]) | Remove single-dimensional entries from the shape of *a*. |
| `std`([axis, dtype, out, ddof, keepdims]) | Returns the standard deviation of the array elements along given axis. |
| `sum`([axis, dtype, out, keepdims, initial, where]) | Return the sum of the array elements over the given axis. |
| `swapaxes`(axis1, axis2) | Return a view of the array with *axis1* and *axis2* interchanged. |
| `take`(indices[, axis, out, mode]) | Return an array formed from the elements of *a* at the given indices. |
| `tobytes`([order]) | Construct Python bytes containing the raw data bytes in the array. |
| `tofile`(fid[, sep, format]) | Write array to a file as text or binary (default). |
| `tolist`() | Return the array as an `a.ndim`-levels deep nested list of Python scalars. |
| `tostring`([order]) | A compatibility alias for `tobytes`, with exactly the same behavior. |
| `trace`([offset, axis1, axis2, dtype, out]) | Return the sum along diagonals of the array. |
| `transpose`(*axes) | Returns a view of the array with axes transposed. |
| `var`([axis, dtype, out, ddof, keepdims]) | Returns the variance of the array elements, along given axis. |
| `view`([dtype][, type]) | New view of array with the same data. |

method

`recarray.`**`all`**`(`*axis=None*, *out=None*, *keepdims=False*`)`

Returns True if all elements evaluate to True.

Refer to `numpy.all` for full documentation.

**See also:**

**`numpy.all`**

equivalent function

method

`recarray.`**`any`**`(`*axis=None*, *out=None*, *keepdims=False*`)`

Returns True if any of the elements of *a* evaluate to True.

Refer to `numpy.any` for full documentation.

**See also:**

**`numpy.any`**

equivalent function

method

`recarray.`**`argmax`**`(`*axis=None*, *out=None*`)`

Return indices of the maximum values along the given axis.

Refer to `numpy.argmax` for full documentation.

**See also:**

***numpy.argmax***

equivalent function

method

`recarray.`**`argmin`**`(`*axis=None*, *out=None*`)`

Return indices of the minimum values along the given axis of *a*.

Refer to `numpy.argmin` for detailed documentation.

**See also:**

***numpy.argmin***

equivalent function

method

`recarray.`**`argpartition`**`(`*kth*, *axis=-1*, *kind='introselect'*, *order=None*`)`

Returns the indices that would partition this array.

Refer to `numpy.argpartition` for full documentation.

New in version 1.8.0.

**See also:**

***numpy.argpartition***

equivalent function

method

`recarray.`**`argsort`**`(`*axis=-1*, *kind=None*, *order=None*`)`

Returns the indices that would sort this array.

Refer to `numpy.argsort` for full documentation.

**See also:**

***numpy.argsort***

equivalent function

method

`recarray.`**`astype`**`(`*dtype*, *order='K'*, *casting='unsafe'*, *subok=True*, *copy=True*`)`

Copy of the array, cast to a specified type.

**Parameters**

**dtype**

[str or dtype] Typecode or data-type to which the array is cast.

**order**

[{'C', 'F', 'A', 'K'}, optional] Controls the memory layout order of the result. 'C' means C order, 'F' means Fortran order, 'A' means 'F' order if all the arrays are Fortran contiguous, 'C' order otherwise, and 'K' means as close to the order the array elements appear in memory as possible. Default is 'K'.

**casting**

[{'no', 'equiv', 'safe', 'same_kind', 'unsafe'}, optional] Controls what kind of data casting may occur. Defaults to 'unsafe' for backwards compatibility.

- 'no' means the data types should not be cast at all.
- 'equiv' means only byte-order changes are allowed.
- 'safe' means only casts which can preserve values are allowed.
- 'same_kind' means only safe casts or casts within a kind, like float64 to float32, are allowed.
- 'unsafe' means any data conversions may be done.

**subok**

[bool, optional] If True, then sub-classes will be passed-through (default), otherwise the returned array will be forced to be a base-class array.

**copy**

[bool, optional] By default, astype always returns a newly allocated array. If this is set to false, and the `dtype`, *order*, and *subok* requirements are satisfied, the input array is returned instead of a copy.

**Returns**

**arr_t**

[ndarray] Unless `copy` is False and the other conditions for returning the input array are satisfied (see description for `copy` input parameter), *arr_t* is a new array of the same shape as the input array, with dtype, order given by `dtype`, *order*.

**Raises**

**ComplexWarning**

When casting from complex to float or int. To avoid this, one should use `a.real.astype(t)`.

### Notes

Changed in version 1.17.0: Casting between a simple data type and a structured one is possible only for "unsafe" casting. Casting to multiple fields is allowed, but casting from multiple fields is not.

Changed in version 1.9.0: Casting from numeric to string types in 'safe' casting mode requires that the string dtype length is long enough to store the max integer/float value converted.

### Examples

```
>>> x = np.array([1, 2, 2.5])
>>> x
array([1. ,  2. ,  2.5])
```

```
>>> x.astype(int)
array([1, 2, 2])
```

method

`recarray.`**`byteswap`**(*inplace=False*)

Swap the bytes of the array elements

Toggle between low-endian and big-endian data representation by returning a byteswapped array, optionally swapped in-place. Arrays of byte-strings are not swapped. The real and imaginary parts of a complex number are swapped individually.

**Parameters**

**inplace**

[bool, optional] If `True`, swap bytes in-place, default is `False`.

**Returns**

**out**

[ndarray] The byteswapped array. If *inplace* is `True`, this is a view to self.

### Examples

```
>>> A = np.array([1, 256, 8755], dtype=np.int16)
>>> list(map(hex, A))
['0x1', '0x100', '0x2233']
>>> A.byteswap(inplace=True)
array([  256,     1, 13090], dtype=int16)
>>> list(map(hex, A))
['0x100', '0x1', '0x3322']
```

Arrays of byte-strings are not swapped

```
>>> A = np.array([b'ceg', b'fac'])
>>> A.byteswap()
array([b'ceg', b'fac'], dtype='|S3')
```

**`A.newbyteorder().byteswap()`** **produces an array with the same values**

but different representation in memory

```
>>> A = np.array([1, 2, 3])
>>> A.view(np.uint8)
array([1, 0, 0, 0, 0, 0, 0, 0, 2, 0, 0, 0, 0, 0, 0, 0, 3, 0, 0, 0, 0, 0,
       0, 0], dtype=uint8)
>>> A.newbyteorder().byteswap(inplace=True)
```

(continues on next page)

(continued from previous page)

```
array([1, 2, 3])
>>> A.view(np.uint8)
array([0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 2, 0, 0, 0, 0, 0, 0,
       0, 3], dtype=uint8)
```

method

`recarray.`**`choose`**`(`*choices*, *out=None*, *mode='raise'*`)`

Use an index array to construct a new array from a set of choices.

Refer to `numpy.choose` for full documentation.

**See also:**

***`numpy.choose`***

equivalent function

method

`recarray.`**`clip`**`(`*min=None*, *max=None*, *out=None*, ***kwargs*`)`

Return an array whose values are limited to `[min, max]`. One of max or min must be given.

Refer to `numpy.clip` for full documentation.

**See also:**

***`numpy.clip`***

equivalent function

method

`recarray.`**`compress`**`(`*condition*, *axis=None*, *out=None*`)`

Return selected slices of this array along given axis.

Refer to `numpy.compress` for full documentation.

**See also:**

***`numpy.compress`***

equivalent function

method

`recarray.`**`conj`**`()`

Complex-conjugate all elements.

Refer to `numpy.conjugate` for full documentation.

**See also:**

***`numpy.conjugate`***

equivalent function

method

`recarray.`**`conjugate`**`()`

Return the complex conjugate, element-wise.

Refer to `numpy.conjugate` for full documentation.

**See also:**

**`numpy.conjugate`**

equivalent function

method

`recarray.`**`copy`**`(`*order='C'*`)`

Return a copy of the array.

**Parameters**

**order**

[{'C', 'F', 'A', 'K'}, optional] Controls the memory layout of the copy. 'C' means C-order, 'F' means F-order, 'A' means 'F' if *a* is Fortran contiguous, 'C' otherwise. 'K' means match the layout of *a* as closely as possible. (Note that this function and `numpy.copy` are very similar, but have different default values for their order= arguments.)

**See also:**

`numpy.copy`, `numpy.copyto`

### Examples

```
>>> x = np.array([[1,2,3],[4,5,6]], order='F')
```

```
>>> y = x.copy()
```

```
>>> x.fill(0)
```

```
>>> x
array([[0, 0, 0],
       [0, 0, 0]])
```

```
>>> y
array([[1, 2, 3],
       [4, 5, 6]])
```

```
>>> y.flags['C_CONTIGUOUS']
True
```

method

`recarray.`**`cumprod`**`(`*axis=None, dtype=None, out=None*`)`

Return the cumulative product of the elements along the given axis.

Refer to `numpy.cumprod` for full documentation.

**See also:**

**`numpy.cumprod`**

equivalent function

method

`recarray.`**`cumsum`**(*axis=None*, *dtype=None*, *out=None*)

Return the cumulative sum of the elements along the given axis.

Refer to `numpy.cumsum` for full documentation.

**See also:**

**`numpy.cumsum`**

equivalent function

method

`recarray.`**`diagonal`**(*offset=0*, *axis1=0*, *axis2=1*)

Return specified diagonals. In NumPy 1.9 the returned array is a read-only view instead of a copy as in previous NumPy versions. In a future version the read-only restriction will be removed.

Refer to `numpy.diagonal` for full documentation.

**See also:**

**`numpy.diagonal`**

equivalent function

method

`recarray.`**`dot`**(*b*, *out=None*)

Dot product of two arrays.

Refer to `numpy.dot` for full documentation.

**See also:**

**`numpy.dot`**

equivalent function

## Examples

```
>>> a = np.eye(2)
>>> b = np.ones((2, 2)) * 2
>>> a.dot(b)
array([[2.,  2.],
       [2.,  2.]])
```

This array method can be conveniently chained:

```
>>> a.dot(b).dot(b)
array([[8.,  8.],
       [8.,  8.]])
```

method

`recarray.`**`dump`**`(`*file*`)`

Dump a pickle of the array to the specified file. The array can be read back with pickle.load or numpy.load.

**Parameters**

**file**

[str or Path] A string naming the dump file.

Changed in version 1.17.0: `pathlib.Path` objects are now accepted.

method

`recarray.`**`dumps`**`()`

Returns the pickle of the array as a string. pickle.loads or numpy.loads will convert the string back to an array.

**Parameters**

**None**

method

`recarray.`**`fill`**`(`*value*`)`

Fill the array with a scalar value.

**Parameters**

**value**

[scalar] All elements of *a* will be assigned this value.

### Examples

```
>>> a = np.array([1, 2])
>>> a.fill(0)
>>> a
array([0, 0])
>>> a = np.empty(2)
>>> a.fill(1)
>>> a
array([1.,  1.])
```

method

`recarray.`**`flatten`**`(`*order='C'*`)`

Return a copy of the array collapsed into one dimension.

**Parameters**

**order**

[{'C', 'F', 'A', 'K'}, optional] 'C' means to flatten in row-major (C-style) order. 'F' means to flatten in column-major (Fortran- style) order. 'A' means to flatten in column-major order if *a* is Fortran *contiguous* in memory, row-major order otherwise. 'K' means to flatten *a* in the order the elements occur in memory. The default is 'C'.

**Returns**

**y**

[ndarray] A copy of the input array, flattened to one dimension.

**See also:**

*`ravel`*

Return a flattened array.

**`flat`**

A 1-D flat iterator over the array.

### Examples

```
>>> a = np.array([[1,2], [3,4]])
>>> a.flatten()
array([1, 2, 3, 4])
>>> a.flatten('F')
array([1, 3, 2, 4])
```

method

`recarray.`**`getfield`**(*dtype*, *offset=0*)

Returns a field of the given array as a certain type.

A field is a view of the array data with a given data-type. The values in the view are determined by the given type and the offset into the current array in bytes. The offset needs to be such that the view dtype fits in the array dtype; for example an array of dtype complex128 has 16-byte elements. If taking a view with a 32-bit integer (4 bytes), the offset needs to be between 0 and 12 bytes.

**Parameters**

**dtype**

[str or dtype] The data type of the view. The dtype size of the view can not be larger than that of the array itself.

**offset**

[int] Number of bytes to skip before beginning the element view.

### Examples

```
>>> x = np.diag([1.+1.j]*2)
>>> x[1, 1] = 2 + 4.j
>>> x
array([[1.+1.j,  0.+0.j],
       [0.+0.j,  2.+4.j]])
>>> x.getfield(np.float64)
array([[1.,  0.],
       [0.,  2.]])
```

By choosing an offset of 8 bytes we can select the complex part of the array for our view:

```
>>> x.getfield(np.float64, offset=8)
array([[1.,  0.],
       [0.,  4.]])
```

method

`recarray.item`(**args*)

Copy an element of an array to a standard Python scalar and return it.

**Parameters**

***args**

[Arguments (variable number and type)]

- none: in this case, the method only works for arrays with one element (*a.size == 1*), which element is copied into a standard Python scalar object and returned.
- int_type: this argument is interpreted as a flat index into the array, specifying which element to copy and return.
- tuple of int_types: functions as does a single int_type argument, except that the argument is interpreted as an nd-index into the array.

**Returns**

**z**

[Standard Python scalar object] A copy of the specified element of the array as a suitable Python scalar

## Notes

When the data type of *a* is longdouble or clongdouble, item() returns a scalar array object because there is no available Python scalar that would not lose information. Void arrays return a buffer object for item(), unless fields are defined, in which case a tuple is returned.

`item` is very similar to a[args], except, instead of an array scalar, a standard Python scalar is returned. This can be useful for speeding up access to elements of the array and doing arithmetic on elements of the array using Python's optimized math.

## Examples

```
>>> np.random.seed(123)
>>> x = np.random.randint(9, size=(3, 3))
>>> x
array([[2, 2, 6],
       [1, 3, 6],
       [1, 0, 1]])
>>> x.item(3)
1
>>> x.item(7)
0
>>> x.item((0, 1))
2
>>> x.item((2, 2))
1
```

method

`recarray.`**`itemset`**`(`**args*`)`

Insert scalar into an array (scalar is cast to array's dtype, if possible)

There must be at least 1 argument, and define the last argument as *item*. Then, `a.itemset(*args)` is equivalent to but faster than `a[args] = item`. The item should be a scalar value and *args* must select a single item in the array *a*.

**Parameters**

***args**

[Arguments] If one argument: a scalar, only used in case *a* is of size 1. If two arguments: the last argument is the value to be set and must be a scalar, the first argument specifies a single array element location. It is either an int or a tuple.

## Notes

Compared to indexing syntax, `itemset` provides some speed increase for placing a scalar into a particular location in an `ndarray`, if you must do this. However, generally this is discouraged: among other problems, it complicates the appearance of the code. Also, when using `itemset` (and `item`) inside a loop, be sure to assign the methods to a local variable to avoid the attribute look-up at each loop iteration.

## Examples

```
>>> np.random.seed(123)
>>> x = np.random.randint(9, size=(3, 3))
>>> x
array([[2, 2, 6],
       [1, 3, 6],
       [1, 0, 1]])
>>> x.itemset(4, 0)
>>> x.itemset((2, 2), 9)
>>> x
array([[2, 2, 6],
       [1, 0, 6],
       [1, 0, 9]])
```

method

`recarray.`**`max`**`(`*axis=None*, *out=None*, *keepdims=False*, *initial=<no value>*, *where=True*`)`

Return the maximum along a given axis.

Refer to `numpy.amax` for full documentation.

**See also:**

**`numpy.amax`**

equivalent function

method

`recarray.`**`mean`**`(`*axis=None*, *dtype=None*, *out=None*, *keepdims=False*`)`

Returns the average of the array elements along given axis.

Refer to `numpy.mean` for full documentation.

**See also:**

**`numpy.mean`**

equivalent function

method

`recarray.`**`min`**`(`*axis=None*, *out=None*, *keepdims=False*, *initial=<no value>*, *where=True*`)`

Return the minimum along a given axis.

Refer to `numpy.amin` for full documentation.

**See also:**

**`numpy.amin`**

equivalent function

method

`recarray.`**`newbyteorder`**`(`*new_order='S'*`)`

Return the array with the same data viewed with a different byte order.

Equivalent to:

```
arr.view(arr.dtype.newbytorder(new_order))
```

Changes are also made in all fields and sub-arrays of the array data type.

**Parameters**

**new_order**

[string, optional] Byte order to force; a value from the byte order specifications below. *new_order* codes can be any of:

- 'S' - swap dtype from current to opposite endian
- {'<', 'L'} - little endian
- {'>', 'B'} - big endian
- {'=', 'N'} - native order
- {'|', 'I'} - ignore (no change to byte order)

The default value ('S') results in swapping the current byte order. The code does a case-insensitive check on the first letter of *new_order* for the alternatives above. For example, any of 'B' or 'b' or 'biggish' are valid to specify big-endian.

**Returns**

**new_arr**

[array] New array object with the dtype reflecting given change to the byte order.

method

`recarray.`**`nonzero`**`()`

Return the indices of the elements that are non-zero.

Refer to `numpy.nonzero` for full documentation.

**See also:**

*`numpy.nonzero`*

equivalent function

method

`recarray`**`.partition`**(*kth*, *axis=-1*, *kind='introselect'*, *order=None*)

Rearranges the elements in the array in such a way that the value of the element in kth position is in the position it would be in a sorted array. All elements smaller than the kth element are moved before this element and all equal or greater are moved behind it. The ordering of the elements in the two partitions is undefined.

New in version 1.8.0.

**Parameters**

**kth**

[int or sequence of ints] Element index to partition by. The kth element value will be in its final sorted position and all smaller elements will be moved before it and all equal or greater elements behind it. The order of all elements in the partitions is undefined. If provided with a sequence of kth it will partition all elements indexed by kth of them into their sorted position at once.

**axis**

[int, optional] Axis along which to sort. Default is -1, which means sort along the last axis.

**kind**

[{'introselect'}, optional] Selection algorithm. Default is 'introselect'.

**order**

[str or list of str, optional] When *a* is an array with fields defined, this argument specifies which fields to compare first, second, etc. A single field can be specified as a string, and not all fields need to be specified, but unspecified fields will still be used, in the order in which they come up in the dtype, to break ties.

**See also:**

*`numpy.partition`*

Return a parititioned copy of an array.

*`argpartition`*

Indirect partition.

*`sort`*

Full sort.

### Notes

See `np.partition` for notes on the different algorithms.

### Examples

```
>>> a = np.array([3, 4, 2, 1])
>>> a.partition(3)
>>> a
array([2, 1, 3, 4])
```

```
>>> a.partition((1, 3))
>>> a
array([1, 2, 3, 4])
```

method

`recarray.`**`prod`**(*axis=None*, *dtype=None*, *out=None*, *keepdims=False*, *initial=1*, *where=True*)

Return the product of the array elements over the given axis

Refer to *`numpy.prod`* for full documentation.

**See also:**

***`numpy.prod`***

equivalent function

method

`recarray.`**`ptp`**(*axis=None*, *out=None*, *keepdims=False*)

Peak to peak (maximum - minimum) value along a given axis.

Refer to *`numpy.ptp`* for full documentation.

**See also:**

***`numpy.ptp`***

equivalent function

method

`recarray.`**`put`**(*indices*, *values*, *mode='raise'*)

Set `a.flat[n] = values[n]` for all *n* in indices.

Refer to *`numpy.put`* for full documentation.

**See also:**

***`numpy.put`***

equivalent function

method

`recarray.`**`ravel`**([*order*])

Return a flattened array.

Refer to *`numpy.ravel`* for full documentation.

**See also:**

**`numpy.ravel`**

equivalent function

**`ndarray.flat`**

a flat iterator on the array.

method

`recarray.`**`repeat`**`(`*repeats*, *axis=None*`)`

Repeat elements of an array.

Refer to `numpy.repeat` for full documentation.

**See also:**

**`numpy.repeat`**

equivalent function

method

`recarray.`**`reshape`**`(`*shape*, *order='C'*`)`

Returns an array containing the same data with a new shape.

Refer to `numpy.reshape` for full documentation.

**See also:**

**`numpy.reshape`**

equivalent function

### Notes

Unlike the free function `numpy.reshape`, this method on `ndarray` allows the elements of the shape parameter to be passed in as separate arguments. For example, `a.reshape(10, 11)` is equivalent to `a.reshape((10, 11))`.

method

`recarray.`**`resize`**`(`*new_shape*, *refcheck=True*`)`

Change shape and size of array in-place.

**Parameters**

**new_shape**

[tuple of ints, or *n* ints] Shape of resized array.

**refcheck**

[bool, optional] If False, reference count will not be checked. Default is True.

**Returns**

**None**

**Raises**

**ValueError**

If *a* does not own its own data or references or views to it exist, and the data memory must be changed. PyPy only: will always raise if the data memory must be changed, since there is no reliable way to determine if references or views to it exist.

**SystemError**

If the *order* keyword argument is specified. This behaviour is a bug in NumPy.

**See also:**

**`resize`**

Return a new array with the specified shape.

## Notes

This reallocates space for the data area if necessary.

Only contiguous arrays (data elements consecutive in memory) can be resized.

The purpose of the reference count check is to make sure you do not use this array as a buffer for another Python object and then reallocate the memory. However, reference counts can increase in other ways so if you are sure that you have not shared the memory for this array with another Python object, then you may safely set *refcheck* to False.

## Examples

Shrinking an array: array is flattened (in the order that the data are stored in memory), resized, and reshaped:

```
>>> a = np.array([[0, 1], [2, 3]], order='C')
>>> a.resize((2, 1))
>>> a
array([[0],
       [1]])
```

```
>>> a = np.array([[0, 1], [2, 3]], order='F')
>>> a.resize((2, 1))
>>> a
array([[0],
       [2]])
```

Enlarging an array: as above, but missing entries are filled with zeros:

```
>>> b = np.array([[0, 1], [2, 3]])
>>> b.resize(2, 3) # new_shape parameter doesn't have to be a tuple
>>> b
array([[0, 1, 2],
       [3, 0, 0]])
```

Referencing an array prevents resizing…

```
>>> c = a
>>> a.resize((1, 1))
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
...
ValueError: cannot resize an array that references or is referenced ...
```

Unless *refcheck* is False:

```
>>> a.resize((1, 1), refcheck=False)
>>> a
array([[0]])
>>> c
array([[0]])
```

method

`recarray.`**`round`**`(`*decimals=0*, *out=None*`)`

Return *a* with each element rounded to the given number of decimals.

Refer to *`numpy.around`* for full documentation.

**See also:**

***`numpy.around`***

equivalent function

method

`recarray.`**`searchsorted`**`(`*v*, *side='left'*, *sorter=None*`)`

Find indices where elements of v should be inserted in a to maintain order.

For full documentation, see *`numpy.searchsorted`*

**See also:**

***`numpy.searchsorted`***

equivalent function

method

`recarray.`**`setfield`**`(`*val*, *dtype*, *offset=0*`)`

Put a value into a specified place in a field defined by a data-type.

Place *val* into *a*'s field defined by *`dtype`* and beginning *offset* bytes into the field.

**Parameters**

**val**

[object] Value to be placed in field.

**dtype**

[dtype object] Data-type of the field in which to place *val*.

**offset**

[int, optional] The number of bytes into the field at which to place *val*.

**Returns**

**None**

**See also:**

*getfield*

### Examples

```
>>> x = np.eye(3)
>>> x.getfield(np.float64)
array([[1.,  0.,  0.],
       [0.,  1.,  0.],
       [0.,  0.,  1.]])
>>> x.setfield(3, np.int32)
>>> x.getfield(np.int32)
array([[3, 3, 3],
       [3, 3, 3],
       [3, 3, 3]], dtype=int32)
>>> x
array([[1.0e+000, 1.5e-323, 1.5e-323],
       [1.5e-323, 1.0e+000, 1.5e-323],
       [1.5e-323, 1.5e-323, 1.0e+000]])
>>> x.setfield(np.eye(3), np.int32)
>>> x
array([[1.,  0.,  0.],
       [0.,  1.,  0.],
       [0.,  0.,  1.]])
```

method

`recarray.`**`setflags`**(*write=None*, *align=None*, *uic=None*)

Set array flags WRITEABLE, ALIGNED, (WRITEBACKIFCOPY and UPDATEIFCOPY), respectively.

These Boolean-valued flags affect how numpy interprets the memory area used by *a* (see Notes below). The ALIGNED flag can only be set to True if the data is actually aligned according to the type. The WRITEBACKIFCOPY and (deprecated) UPDATEIFCOPY flags can never be set to True. The flag WRITEABLE can only be set to True if the array owns its own memory, or the ultimate owner of the memory exposes a writeable buffer interface, or is a string. (The exception for string is made so that unpickling can be done without copying memory.)

**Parameters**

**write**

[bool, optional] Describes whether or not *a* can be written to.

**align**

[bool, optional] Describes whether or not *a* is aligned properly for its type.

**uic**

[bool, optional] Describes whether or not *a* is a copy of another "base" array.

### Notes

Array flags provide information about how the memory area used for the array is to be interpreted. There are 7 Boolean flags in use, only four of which can be changed by the user: WRITEBACKIFCOPY, UPDATEIFCOPY, WRITEABLE, and ALIGNED.

WRITEABLE (W) the data area can be written to;

ALIGNED (A) the data and strides are aligned appropriately for the hardware (as determined by the compiler);

UPDATEIFCOPY (U) (deprecated), replaced by WRITEBACKIFCOPY;

WRITEBACKIFCOPY (X) this array is a copy of some other array (referenced by .base). When the C-API function PyArray_ResolveWritebackIfCopy is called, the base array will be updated with the contents of this array.

All flags can be accessed using the single (upper case) letter as well as the full name.

### Examples

```
>>> y = np.array([[3, 1, 7],
...               [2, 0, 0],
...               [8, 5, 9]])
>>> y
array([[3, 1, 7],
       [2, 0, 0],
       [8, 5, 9]])
>>> y.flags
  C_CONTIGUOUS : True
  F_CONTIGUOUS : False
  OWNDATA : True
  WRITEABLE : True
  ALIGNED : True
  WRITEBACKIFCOPY : False
  UPDATEIFCOPY : False
>>> y.setflags(write=0, align=0)
>>> y.flags
  C_CONTIGUOUS : True
  F_CONTIGUOUS : False
  OWNDATA : True
  WRITEABLE : False
  ALIGNED : False
  WRITEBACKIFCOPY : False
  UPDATEIFCOPY : False
>>> y.setflags(uic=1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: cannot set WRITEBACKIFCOPY flag to True
```

method

`recarray.`**`sort`**`(`*axis=-1*, *kind=None*, *order=None*`)`

Sort an array in-place. Refer to `numpy.sort` for full documentation.

**Parameters**

**axis**

[int, optional] Axis along which to sort. Default is -1, which means sort along the last axis.

**kind**

[{'quicksort', 'mergesort', 'heapsort', 'stable'}, optional] Sorting algorithm. The default is 'quicksort'. Note that both 'stable' and 'mergesort' use timsort under the covers and, in general, the actual implementation will vary with datatype. The 'mergesort' option is retained for backwards compatibility.

Changed in version 1.15.0.: The 'stable' option was added.

**order**

[str or list of str, optional] When *a* is an array with fields defined, this argument specifies which fields to compare first, second, etc. A single field can be specified as a string, and not all fields need be specified, but unspecified fields will still be used, in the order in which they come up in the dtype, to break ties.

**See also:**

**`numpy.sort`**

Return a sorted copy of an array.

**`numpy.argsort`**

Indirect sort.

**`numpy.lexsort`**

Indirect stable sort on multiple keys.

**`numpy.searchsorted`**

Find elements in sorted array.

**`numpy.partition`**

Partial sort.

### Notes

See `numpy.sort` for notes on the different sorting algorithms.

### Examples

```
>>> a = np.array([[1,4], [3,1]])
>>> a.sort(axis=1)
>>> a
array([[1, 4],
       [1, 3]])
>>> a.sort(axis=0)
>>> a
array([[1, 3],
       [1, 4]])
```

Use the *order* keyword to specify a field to use when sorting a structured array:

```
>>> a = np.array([('a', 2), ('c', 1)], dtype=[('x', 'S1'), ('y', int)])
>>> a.sort(order='y')
>>> a
array([(b'c', 1), (b'a', 2)],
      dtype=[('x', 'S1'), ('y', '<i8')])
```

method

`recarray.`**`squeeze`**(*axis=None*)

Remove single-dimensional entries from the shape of *a*.

Refer to `numpy.squeeze` for full documentation.

**See also:**

**`numpy.squeeze`**

equivalent function

method

`recarray.`**`std`**(*axis=None*, *dtype=None*, *out=None*, *ddof=0*, *keepdims=False*)

Returns the standard deviation of the array elements along given axis.

Refer to `numpy.std` for full documentation.

**See also:**

**`numpy.std`**

equivalent function

method

`recarray.`**`sum`**(*axis=None*, *dtype=None*, *out=None*, *keepdims=False*, *initial=0*, *where=True*)

Return the sum of the array elements over the given axis.

Refer to `numpy.sum` for full documentation.

**See also:**

**`numpy.sum`**

equivalent function

method

`recarray.`**`swapaxes`**(*axis1*, *axis2*)

Return a view of the array with *axis1* and *axis2* interchanged.

Refer to `numpy.swapaxes` for full documentation.

**See also:**

**`numpy.swapaxes`**

equivalent function

method

`recarray.`**`take`**`(`*indices*, *axis=None*, *out=None*, *mode='raise'*`)`

Return an array formed from the elements of *a* at the given indices.

Refer to `numpy.take` for full documentation.

**See also:**

**`numpy.take`**

equivalent function

method

`recarray.`**`tobytes`**`(`*order='C'*`)`

Construct Python bytes containing the raw data bytes in the array.

Constructs Python bytes showing a copy of the raw contents of data memory. The bytes object can be produced in either 'C' or 'Fortran', or 'Any' order (the default is 'C'-order). 'Any' order means C-order unless the F_CONTIGUOUS flag in the array is set, in which case it means 'Fortran' order.

New in version 1.9.0.

**Parameters**

**order**

[{'C', 'F', None}, optional] Order of the data for multidimensional arrays: C, Fortran, or the same as for the original array.

**Returns**

**s**

[bytes] Python bytes exhibiting a copy of *a*'s raw data.

## Examples

```
>>> x = np.array([[0, 1], [2, 3]], dtype='<u2')
>>> x.tobytes()
b'\x00\x00\x01\x00\x02\x00\x03\x00'
>>> x.tobytes('C') == x.tobytes()
True
>>> x.tobytes('F')
b'\x00\x00\x02\x00\x01\x00\x03\x00'
```

method

`recarray.`**`tofile`**`(`*fid*, *sep=""*, *format="%s"*`)`

Write array to a file as text or binary (default).

Data is always written in 'C' order, independent of the order of *a*. The data produced by this method can be recovered using the function fromfile().

**Parameters**

**fid**

[file or str or Path] An open file object, or a string containing a filename.

Changed in version 1.17.0: `pathlib.Path` objects are now accepted.

**sep**

[str] Separator between array items for text output. If "" (empty), a binary file is written, equivalent to `file.write(a.tobytes())`.

**format**

[str] Format string for text file output. Each entry in the array is formatted to text by first converting it to the closest Python type, and then using "format" % item.

## Notes

This is a convenience function for quick storage of array data. Information on endianness and precision is lost, so this method is not a good choice for files intended to archive data or transport data between machines with different endianness. Some of these problems can be overcome by outputting the data as text files, at the expense of speed and file size.

When fid is a file object, array contents are directly written to the file, bypassing the file object's `write` method. As a result, tofile cannot be used with files objects supporting compression (e.g., GzipFile) or file-like objects that do not support `fileno()` (e.g., BytesIO).

method

`recarray.`**`tolist`**`()`

Return the array as an `a.ndim`-levels deep nested list of Python scalars.

Return a copy of the array data as a (nested) Python list. Data items are converted to the nearest compatible builtin Python type, via the *`item`* function.

If `a.ndim` is 0, then since the depth of the nested list is 0, it will not be a list at all, but a simple Python scalar.

**Parameters**

**none**

**Returns**

**y**

[object, or list of object, or list of list of object, or …] The possibly nested list of array elements.

## Notes

The array may be recreated via `a = np.array(a.tolist())`, although this may sometimes lose precision.

### Examples

For a 1D array, `a.tolist()` is almost the same as `list(a)`, except that `tolist` changes numpy scalars to Python scalars:

```
>>> a = np.uint32([1, 2])
>>> a_list = list(a)
>>> a_list
[1, 2]
>>> type(a_list[0])
<class 'numpy.uint32'>
>>> a_tolist = a.tolist()
>>> a_tolist
[1, 2]
>>> type(a_tolist[0])
<class 'int'>
```

Additionally, for a 2D array, `tolist` applies recursively:

```
>>> a = np.array([[1, 2], [3, 4]])
>>> list(a)
[array([1, 2]), array([3, 4])]
>>> a.tolist()
[[1, 2], [3, 4]]
```

The base case for this recursion is a 0D array:

```
>>> a = np.array(1)
>>> list(a)
Traceback (most recent call last):
  ...
TypeError: iteration over a 0-d array
>>> a.tolist()
1
```

method

`recarray.`**`tostring`**(*order='C'*)

A compatibility alias for *`tobytes`*, with exactly the same behavior.

Despite its name, it returns *bytes* not `str`s.

Deprecated since version 1.19.0.

method

`recarray.`**`trace`**(*offset=0*, *axis1=0*, *axis2=1*, *dtype=None*, *out=None*)

Return the sum along diagonals of the array.

Refer to *`numpy.trace`* for full documentation.

**See also:**

***`numpy.trace`***

equivalent function

method

`recarray.`**`transpose`**(**axes*)

Returns a view of the array with axes transposed.

For a 1-D array this has no effect, as a transposed vector is simply the same vector. To convert a 1-D array into a 2D column vector, an additional dimension must be added. *np.atleast2d(a).T* achieves this, as does *a[:, np.newaxis]*. For a 2-D array, this is a standard matrix transpose. For an n-D array, if axes are given, their order indicates how the axes are permuted (see Examples). If axes are not provided and `a.shape = (i[0], i[1], ... i[n-2], i[n-1])`, then `a.transpose().shape = (i[n-1], i[n-2], ... i[1], i[0])`.

**Parameters**

**axes**

[None, tuple of ints, or *n* ints]

- None or no argument: reverses the order of the axes.
- tuple of ints: *i* in the *j*-th place in the tuple means *a*'s *i*-th axis becomes *a.transpose()*'s *j*-th axis.
- *n* ints: same as an n-tuple of the same ints (this form is intended simply as a "convenience" alternative to the tuple form)

**Returns**

**out**

[ndarray] View of *a*, with axes suitably permuted.

**See also:**

**`ndarray.T`**

Array property returning the array transposed.

**`ndarray.reshape`**

Give a new shape to an array without changing its data.

## Examples

```
>>> a = np.array([[1, 2], [3, 4]])
>>> a
array([[1, 2],
       [3, 4]])
>>> a.transpose()
array([[1, 3],
       [2, 4]])
>>> a.transpose((1, 0))
array([[1, 3],
       [2, 4]])
>>> a.transpose(1, 0)
array([[1, 3],
       [2, 4]])
```

method

`recarray.`**`var`**(*axis=None*, *dtype=None*, *out=None*, *ddof=0*, *keepdims=False*)

Returns the variance of the array elements, along given axis.

Refer to `numpy.var` for full documentation.

**See also:**

[`numpy.var`]

equivalent function

method

`recarray.`**`view`**`(`*[dtype][, type]*`)`

New view of array with the same data.

---

**Note:** Passing None for `dtype` is different from omitting the parameter, since the former invokes `dtype(None)` which is an alias for `dtype('float_')`.

---

**Parameters**

**dtype**

[data-type or ndarray sub-class, optional] Data-type descriptor of the returned view, e.g., float32 or int16. Omitting it results in the view having the same data-type as *a*. This argument can also be specified as an ndarray sub-class, which then specifies the type of the returned object (this is equivalent to setting the `type` parameter).

**type**

[Python type, optional] Type of the returned view, e.g., ndarray or matrix. Again, omission of the parameter results in type preservation.

### Notes

`a.view()` is used two different ways:

`a.view(some_dtype)` or `a.view(dtype=some_dtype)` constructs a view of the array's memory with a different data-type. This can cause a reinterpretation of the bytes of memory.

`a.view(ndarray_subclass)` or `a.view(type=ndarray_subclass)` just returns an instance of *ndarray_subclass* that looks at the same array (same shape, dtype, etc.) This does not cause a reinterpretation of the memory.

For `a.view(some_dtype)`, if `some_dtype` has a different number of bytes per entry than the previous dtype (for example, converting a regular array to a structured array), then the behavior of the view cannot be predicted just from the superficial appearance of `a` (shown by `print(a)`). It also depends on exactly how `a` is stored in memory. Therefore if `a` is C-ordered versus fortran-ordered, versus defined as a slice or transpose, etc., the view may give different results.

### Examples

```
>>> x = np.array([(1, 2)], dtype=[('a', np.int8), ('b', np.int8)])
```

Viewing array data using a different type and dtype:

```
>>> y = x.view(dtype=np.int16, type=np.matrix)
>>> y
matrix([[513]], dtype=int16)
>>> print(type(y))
<class 'numpy.matrix'>
```

Creating a view on a structured array so it can be used in calculations

```
>>> x = np.array([(1, 2),(3,4)], dtype=[('a', np.int8), ('b', np.int8)])
>>> xv = x.view(dtype=np.int8).reshape(-1,2)
>>> xv
array([[1, 2],
       [3, 4]], dtype=int8)
>>> xv.mean(0)
array([2.,  3.])
```

Making changes to the view changes the underlying array

```
>>> xv[0,1] = 20
>>> x
array([(1, 20), (3,  4)], dtype=[('a', 'i1'), ('b', 'i1')])
```

Using a view to convert an array to a recarray:

```
>>> z = x.view(np.recarray)
>>> z.a
array([1, 3], dtype=int8)
```

Views share data:

```
>>> x[0] = (9, 10)
>>> z[0]
(9, 10)
```

Views that change the dtype size (bytes per entry) should normally be avoided on arrays defined by slices, transposes, fortran-ordering, etc.:

```
>>> x = np.array([[1,2,3],[4,5,6]], dtype=np.int16)
>>> y = x[:, 0:2]
>>> y
array([[1, 2],
       [4, 5]], dtype=int16)
>>> y.view(dtype=[('width', np.int16), ('length', np.int16)])
Traceback (most recent call last):
    ...
ValueError: To change to a dtype of a different size, the array must be C-
↪contiguous
>>> z = y.copy()
>>> z.view(dtype=[('width', np.int16), ('length', np.int16)])
array([[(1, 2)],
       [(4, 5)]], dtype=[('width', '<i2'), ('length', '<i2')])
```

| **field** | |
|---|---|

**class** `numpy.`**`record`**

A data-type scalar that allows field access as attribute lookup.

**Attributes**

**`T`**

transpose

**`base`**

base object

**`data`**

pointer to start of data

*`dtype`*

dtype object

**`flags`**

integer value of flags

**`flat`**

a 1-d view of scalar

*`imag`*

imaginary part of scalar

**`itemsize`**

length of one element in bytes

**`nbytes`**

length of item in bytes

**`ndim`**

number of array dimensions

*`real`*

real part of scalar

*`shape`*

tuple of array dimensions

**`size`**

number of elements in the gentype

**`strides`**

tuple of bytes steps in each dimension

### Methods

| | |
|---|---|
| `all`() | Not implemented (virtual attribute) |
| `any`() | Not implemented (virtual attribute) |
| `argmax`() | Not implemented (virtual attribute) |
| `argmin`() | Not implemented (virtual attribute) |
| `argsort`() | Not implemented (virtual attribute) |
| `astype`() | Not implemented (virtual attribute) |
| `byteswap`() | Not implemented (virtual attribute) |
| `choose`() | Not implemented (virtual attribute) |
| `clip`() | Not implemented (virtual attribute) |
| `compress`() | Not implemented (virtual attribute) |
| `conjugate`() | Not implemented (virtual attribute) |
| `copy`() | Not implemented (virtual attribute) |
| `cumprod`() | Not implemented (virtual attribute) |
| `cumsum`() | Not implemented (virtual attribute) |
| `diagonal`() | Not implemented (virtual attribute) |
| `dump`() | Not implemented (virtual attribute) |
| `dumps`() | Not implemented (virtual attribute) |
| `fill`() | Not implemented (virtual attribute) |
| `flatten`() | Not implemented (virtual attribute) |
| `getfield`() | Not implemented (virtual attribute) |
| `item`() | Not implemented (virtual attribute) |
| `itemset`() | Not implemented (virtual attribute) |
| `max`() | Not implemented (virtual attribute) |
| `mean`() | Not implemented (virtual attribute) |
| `min`() | Not implemented (virtual attribute) |
| `newbyteorder`([new_order]) | Return a new `dtype` with a different byte order. |
| `nonzero`() | Not implemented (virtual attribute) |
| `pprint`(self) | Pretty-print all fields. |
| `prod`() | Not implemented (virtual attribute) |
| `ptp`() | Not implemented (virtual attribute) |
| `put`() | Not implemented (virtual attribute) |
| `ravel`() | Not implemented (virtual attribute) |
| `repeat`() | Not implemented (virtual attribute) |
| `reshape`() | Not implemented (virtual attribute) |
| `resize`() | Not implemented (virtual attribute) |
| `round`() | Not implemented (virtual attribute) |
| `searchsorted`() | Not implemented (virtual attribute) |
| `setfield`() | Not implemented (virtual attribute) |
| `setflags`() | Not implemented (virtual attribute) |
| `sort`() | Not implemented (virtual attribute) |
| `squeeze`() | Not implemented (virtual attribute) |
| `std`() | Not implemented (virtual attribute) |
| `sum`() | Not implemented (virtual attribute) |
| `swapaxes`() | Not implemented (virtual attribute) |
| `take`() | Not implemented (virtual attribute) |
| `tofile`() | Not implemented (virtual attribute) |
| `tolist`() | Not implemented (virtual attribute) |
| `tostring`() | Not implemented (virtual attribute) |

Continued on next page

Table 44 – continued from previous page

| | |
|---|---|
| *trace*() | Not implemented (virtual attribute) |
| *transpose*() | Not implemented (virtual attribute) |
| *var*() | Not implemented (virtual attribute) |
| *view*() | Not implemented (virtual attribute) |

method

`record.all()`

Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also the corresponding attribute of the derived class of interest.

method

`record.any()`

Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also the corresponding attribute of the derived class of interest.

method

`record.argmax()`

Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also the corresponding attribute of the derived class of interest.

method

`record.argmin()`

Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also the corresponding attribute of the derived class of interest.

method

`record.argsort()`

Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also the corresponding attribute of the derived class of interest.

method

`record.astype()`

Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also the corresponding attribute of the derived class of interest.

method

`record.`**`byteswap`**`()`

Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also the corresponding attribute of the derived class of interest.

method

`record.`**`choose`**`()`

Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also the corresponding attribute of the derived class of interest.

method

`record.`**`clip`**`()`

Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also the corresponding attribute of the derived class of interest.

method

`record.`**`compress`**`()`

Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also the corresponding attribute of the derived class of interest.

method

`record.`**`conjugate`**`()`

Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also the corresponding attribute of the derived class of interest.

method

`record.`**`copy`**`()`

Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also the corresponding attribute of the derived class of interest.

method

`record.`**`cumprod`**`()`

Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also the corresponding attribute of the derived class of interest.

method

`record.`**`cumsum`**`()`

Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also the corresponding attribute of the derived class of interest.

method

`record.`**`diagonal`**`()`

Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also the corresponding attribute of the derived class of interest.

method

`record.`**`dump`**`()`

Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also the corresponding attribute of the derived class of interest.

method

`record.`**`dumps`**`()`

Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also the corresponding attribute of the derived class of interest.

method

`record.`**`fill`**`()`

Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also the corresponding attribute of the derived class of interest.

method

`record.`**`flatten`**`()`

Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also the corresponding attribute of the derived class of interest.

method

`record.`**`getfield`**`()`

Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also the corresponding attribute of the derived class of interest.

method

`record.`**`item`**`()`

Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also the corresponding attribute of the derived class of interest.

method

`record.`**`itemset`**`()`

Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also the corresponding attribute of the derived class of interest.

method

`record.`**`max`**`()`

Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also the corresponding attribute of the derived class of interest.

method

`record.`**`mean`**`()`

Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also the corresponding attribute of the derived class of interest.

method

`record.`**`min`**`()`

Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also the corresponding attribute of the derived class of interest.

method

`record.`**`newbyteorder`**`(`*new_order='S'*`)`

Return a new `dtype` with a different byte order.

Changes are also made in all fields and sub-arrays of the data type.

The *new_order* code can be any from the following:

- 'S' - swap dtype from current to opposite endian
- {'<', 'L'} - little endian

- {'>', 'B'} - big endian
- {'=', 'N'} - native order
- {'|', 'I'} - ignore (no change to byte order)

**Parameters**

**new_order**

[str, optional] Byte order to force; a value from the byte order specifications above. The default value ('S') results in swapping the current byte order. The code does a case-insensitive check on the first letter of *new_order* for the alternatives above. For example, any of 'B' or 'b' or 'biggish' are valid to specify big-endian.

**Returns**

**new_dtype**

[dtype] New `dtype` object with the given change to the byte order.

method

`record.`**`nonzero`**`()`

Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also the corresponding attribute of the derived class of interest.

method

`record.`**`pprint`**`(`*self*`)`

Pretty-print all fields.

method

`record.`**`prod`**`()`

Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also the corresponding attribute of the derived class of interest.

method

`record.`**`ptp`**`()`

Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also the corresponding attribute of the derived class of interest.

method

`record.`**`put`**`()`

Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also the corresponding attribute of the derived class of interest.

method

`record.`**`ravel`**`()`

Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also the corresponding attribute of the derived class of interest.

method

`record.`**`repeat`**`()`

Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also the corresponding attribute of the derived class of interest.

method

`record.`**`reshape`**`()`

Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also the corresponding attribute of the derived class of interest.

method

`record.`**`resize`**`()`

Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also the corresponding attribute of the derived class of interest.

method

`record.`**`round`**`()`

Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also the corresponding attribute of the derived class of interest.

method

`record.`**`searchsorted`**`()`

Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also the corresponding attribute of the derived class of interest.

method

`record.`**`setfield`**`()`

Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also the corresponding attribute of the derived class of interest.

method

`record.`**`setflags`**`()`

Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also the corresponding attribute of the derived class of interest.

method

`record.`**`sort`**`()`

Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also the corresponding attribute of the derived class of interest.

method

`record.`**`squeeze`**`()`

Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also the corresponding attribute of the derived class of interest.

method

`record.`**`std`**`()`

Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also the corresponding attribute of the derived class of interest.

method

`record.`**`sum`**`()`

Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also the corresponding attribute of the derived class of interest.

method

`record.`**`swapaxes`**`()`

Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also the corresponding attribute of the derived class of interest.

method

`record.take()`

Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also the corresponding attribute of the derived class of interest.

method

`record.tofile()`

Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also the corresponding attribute of the derived class of interest.

method

`record.tolist()`

Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also the corresponding attribute of the derived class of interest.

method

`record.tostring()`

Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also the corresponding attribute of the derived class of interest.

method

`record.trace()`

Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also the corresponding attribute of the derived class of interest.

method

`record.transpose()`

Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also the corresponding attribute of the derived class of interest.

method

`record.var()`

Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also the corresponding attribute of the derived class of interest.

method

`record.`**`view`**`()`

Not implemented (virtual attribute)

Class generic exists solely to derive numpy scalars from, and possesses, albeit unimplemented, all the attributes of the ndarray class so as to provide a uniform API.

See also the corresponding attribute of the derived class of interest.

| | |
|---|---|
| **conj** | |
| **tobytes** | |

## 1.6.6 Masked arrays (`numpy.ma`)

**See also:**

*Masked arrays*

## 1.6.7 Standard container class

For backward compatibility and as a standard "container "class, the UserArray from Numeric has been brought over to NumPy and named *`numpy.lib.user_array.container`* The container class is a Python class whose self.array attribute is an ndarray. Multiple inheritance is probably easier with numpy.lib.user_array.container than with the ndarray itself and so it is included by default. It is not documented here beyond mentioning its existence because you are encouraged to use the ndarray class directly if you can.

| | |
|---|---|
| *`numpy.lib.user_array.container`*(data[, …]) | Standard container-class for easy multiple-inheritance. |

**`class`** `numpy.lib.user_array.`**`container`**`(`*data*, *dtype=None*, *copy=True*`)`

Standard container-class for easy multiple-inheritance.

### Methods

| | |
|---|---|
| **copy** | |
| **tostring** | |
| **byteswap** | |
| **astype** | |

## 1.6.8 Array Iterators

Iterators are a powerful concept for array processing. Essentially, iterators implement a generalized for-loop. If *myiter* is an iterator object, then the Python code:

```
for val in myiter:
    ...
    some code involving val
    ...
```

calls `val = next(myiter)` repeatedly until `StopIteration` is raised by the iterator. There are several ways to iterate over an array that may be useful: default iteration, flat iteration, and $N$-dimensional enumeration.

### Default iteration

The default iterator of an ndarray object is the default Python iterator of a sequence type. Thus, when the array object itself is used as an iterator. The default behavior is equivalent to:

```
for i in range(arr.shape[0]):
    val = arr[i]
```

This default iterator selects a sub-array of dimension $N-1$ from the array. This can be a useful construct for defining recursive algorithms. To loop over the entire array requires $N$ for-loops.

```
>>> a = np.arange(24).reshape(3,2,4)+10
>>> for val in a:
...     print('item:', val)
item: [[10 11 12 13]
 [14 15 16 17]]
item: [[18 19 20 21]
 [22 23 24 25]]
item: [[26 27 28 29]
 [30 31 32 33]]
```

### Flat iteration

| | |
|---|---|
| *`ndarray.flat`* | A 1-D iterator over the array. |

As mentioned previously, the flat attribute of ndarray objects returns an iterator that will cycle over the entire array in C-style contiguous order.

```
>>> for i, val in enumerate(a.flat):
...     if i%5 == 0: print(i, val)
0 10
5 15
10 20
15 25
20 30
```

Here, I've used the built-in enumerate iterator to return the iterator index as well as the value.

### N-dimensional enumeration

| | |
|---|---|
| *`ndenumerate`*(arr) | Multidimensional index iterator. |

**class** `numpy.`**`ndenumerate`**(*arr*)

Multidimensional index iterator.

Return an iterator yielding pairs of array coordinates and values.

**Parameters**

**arr**

[ndarray] Input array.

**See also:**

*ndindex*, *flatiter*

### Examples

```
>>> a = np.array([[1, 2], [3, 4]])
>>> for index, x in np.ndenumerate(a):
...     print(index, x)
(0, 0) 1
(0, 1) 2
(1, 0) 3
(1, 1) 4
```

Sometimes it may be useful to get the N-dimensional index while iterating. The ndenumerate iterator can achieve this.

```
>>> for i, val in np.ndenumerate(a):
...     if sum(i)%5 == 0: print(i, val)
(0, 0, 0) 10
(1, 1, 3) 25
(2, 0, 3) 29
(2, 1, 2) 32
```

## Iterator for broadcasting

| | |
|---|---|
| *broadcast* | Produce an object that mimics broadcasting. |

**class** `numpy.`**`broadcast`**

Produce an object that mimics broadcasting.

**Parameters**

**in1, in2, …**

[array_like] Input parameters.

**Returns**

**b**

[broadcast object] Broadcast the input parameters against one another, and return an object that encapsulates the result. Amongst others, it has `shape` and `nd` properties, and may be used as an iterator.

**See also:**

*broadcast_arrays*, *broadcast_to*

## Examples

Manually adding two vectors, using broadcasting:

```
>>> x = np.array([[1], [2], [3]])
>>> y = np.array([4, 5, 6])
>>> b = np.broadcast(x, y)
```

```
>>> out = np.empty(b.shape)
>>> out.flat = [u+v for (u,v) in b]
>>> out
array([[5.,  6.,  7.],
       [6.,  7.,  8.],
       [7.,  8.,  9.]])
```

Compare against built-in broadcasting:

```
>>> x + y
array([[5, 6, 7],
       [6, 7, 8],
       [7, 8, 9]])
```

### Attributes

**index**

current index in broadcasted result

**iters**

tuple of iterators along `self`'s "components."

**nd**

Number of dimensions of broadcasted result.

**ndim**

Number of dimensions of broadcasted result.

**numiter**

Number of iterators possessed by the broadcasted result.

***shape***

Shape of broadcasted result.

**size**

Total size of broadcasted result.

### Methods

| | |
|---|---|
| `reset()` | Reset the broadcasted result's iterator(s). |

method

`broadcast.`**`reset`**`()`

Reset the broadcasted result's iterator(s).

**Parameters**

**None**

**Returns**

**None**

### Examples

```
>>> x = np.array([1, 2, 3])
>>> y = np.array([[4], [5], [6]])
>>> b = np.broadcast(x, y)
>>> b.index
0
>>> next(b), next(b), next(b)
((1, 4), (2, 4), (3, 4))
>>> b.index
3
>>> b.reset()
>>> b.index
0
```

The general concept of broadcasting is also available from Python using the `broadcast` iterator. This object takes $N$ objects as inputs and returns an iterator that returns tuples providing each of the input sequence elements in the broadcasted result.

```
>>> for val in np.broadcast([[1,0],[2,3]],[0,1]):
...     print(val)
(1, 0)
(0, 1)
(2, 0)
(3, 1)
```

## 1.7 Masked arrays

Masked arrays are arrays that may have missing or invalid entries. The `numpy.ma` module provides a nearly work-alike replacement for numpy that supports data arrays with masks.

### 1.7.1 The `numpy.ma` module

#### Rationale

Masked arrays are arrays that may have missing or invalid entries. The *numpy.ma* module provides a nearly work-alike replacement for numpy that supports data arrays with masks.

#### What is a masked array?

In many circumstances, datasets can be incomplete or tainted by the presence of invalid data. For example, a sensor may have failed to record a data, or recorded an invalid value. The *numpy.ma* module provides a convenient way to address this issue, by introducing masked arrays.

A masked array is the combination of a standard *numpy.ndarray* and a mask. A mask is either *nomask*, indicating that no value of the associated array is invalid, or an array of booleans that determines for each element of the associated array whether the value is valid or not. When an element of the mask is `False`, the corresponding element of the associated array is valid and is said to be unmasked. When an element of the mask is `True`, the corresponding element of the associated array is said to be masked (invalid).

The package ensures that masked entries are not used in computations.

As an illustration, let's consider the following dataset:

```
>>> import numpy as np
>>> import numpy.ma as ma
>>> x = np.array([1, 2, 3, -1, 5])
```

We wish to mark the fourth entry as invalid. The easiest is to create a masked array:

```
>>> mx = ma.masked_array(x, mask=[0, 0, 0, 1, 0])
```

We can now compute the mean of the dataset, without taking the invalid data into account:

```
>>> mx.mean()
2.75
```

#### The `numpy.ma` module

The main feature of the *numpy.ma* module is the *MaskedArray* class, which is a subclass of *numpy.ndarray*. The class, its attributes and methods are described in more details in the *MaskedArray class* section.

The *numpy.ma* module can be used as an addition to *numpy*:

```
>>> import numpy as np
>>> import numpy.ma as ma
```

To create an array with the second element invalid, we would do:

```
>>> y = ma.array([1, 2, 3], mask = [0, 1, 0])
```

To create a masked array where all values close to 1.e20 are invalid, we would do:

```
>>> z = ma.masked_values([1.0, 1.e20, 3.0, 4.0], 1.e20)
```

For a complete discussion of creation methods for masked arrays please see section *Constructing masked arrays*.

## 1.7.2 Using numpy.ma

### Constructing masked arrays

There are several ways to construct a masked array.

- A first possibility is to directly invoke the `MaskedArray` class.
- A second possibility is to use the two masked array constructors, `array` and `masked_array`.

| | |
|---|---|
| `array`(data[, dtype, copy, order, mask, …]) | An array class with possibly masked values. |
| `masked_array` | alias of `numpy.ma.core.MaskedArray` |

`numpy.ma.`**`array`**`(`*data*, *dtype=None*, *copy=False*, *order=None*, *mask=False*, *fill_value=None*, *keep_mask=True*, *hard_mask=False*, *shrink=True*, *subok=True*, *ndmin=0*`)`

An array class with possibly masked values.

Masked values of True exclude the corresponding element from any computation.

Construction:

```
x = MaskedArray(data, mask=nomask, dtype=None, copy=False, subok=True,
                ndmin=0, fill_value=None, keep_mask=True, hard_mask=None,
                shrink=True, order=None)
```

**Parameters**

**data**

[array_like] Input data.

**mask**

[sequence, optional] Mask. Must be convertible to an array of booleans with the same shape as *data*. True indicates a masked (i.e. invalid) data.

**dtype**

[dtype, optional] Data type of the output. If *dtype* is None, the type of the data argument (`data.dtype`) is used. If *dtype* is not None and different from `data.dtype`, a copy is performed.

**copy**

[bool, optional] Whether to copy the input data (True), or to use a reference instead. Default is False.

**subok**

[bool, optional] Whether to return a subclass of `MaskedArray` if possible (True) or a plain `MaskedArray`. Default is True.

**ndmin**

[int, optional] Minimum number of dimensions. Default is 0.

**fill_value**

[scalar, optional] Value used to fill in the masked values when necessary. If None, a default based on the data-type is used.

**keep_mask**

[bool, optional] Whether to combine *mask* with the mask of the input data, if any (True), or to use only *mask* for the output (False). Default is True.

**hard_mask**

[bool, optional] Whether to use a hard mask or not. With a hard mask, masked values cannot be unmasked. Default is False.

**shrink**

[bool, optional] Whether to force compression of an empty mask. Default is True.

**order**

[{'C', 'F', 'A'}, optional] Specify the order of the array. If order is 'C', then the array will be in C-contiguous order (last-index varies the fastest). If order is 'F', then the returned array will be in Fortran-contiguous order (first-index varies the fastest). If order is 'A' (default), then the returned array may be in any order (either C-, Fortran-contiguous, or even discontiguous), unless a copy is required, in which case it will be C-contiguous.

## Examples

The `mask` can be initialized with an array of boolean values with the same shape as `data`.

```
>>> data = np.arange(6).reshape((2, 3))
>>> np.ma.MaskedArray(data, mask=[[False, True, False],
...                                [False, False, True]])
masked_array(
  data=[[0, --, 2],
        [3, 4, --]],
  mask=[[False,  True, False],
        [False, False,  True]],
  fill_value=999999)
```

Alternatively, the `mask` can be initialized to homogeneous boolean array with the same shape as `data` by passing in a scalar boolean value:

```
>>> np.ma.MaskedArray(data, mask=False)
masked_array(
  data=[[0, 1, 2],
        [3, 4, 5]],
  mask=[[False, False, False],
        [False, False, False]],
  fill_value=999999)
```

```
>>> np.ma.MaskedArray(data, mask=True)
masked_array(
  data=[[--, --, --],
        [--, --, --]],
  mask=[[ True,  True,  True],
        [ True,  True,  True]],
  fill_value=999999,
  dtype=int64)
```

---

**Note:** The recommended practice for initializing `mask` with a scalar boolean value is to use `True`/`False` rather than `np.True_`/`np.False_`. The reason is *`nomask`* is represented internally as `np.False_`.

```
>>> np.False_ is np.ma.nomask
True
```

---

`numpy.ma.`**`masked_array`**

alias of `numpy.ma.core.MaskedArray`

- A third option is to take the view of an existing array. In that case, the mask of the view is set to *`nomask`* if the array has no named fields, or an array of boolean with the same structure as the array otherwise.

```
>>> x = np.array([1, 2, 3])
>>> x.view(ma.MaskedArray)
masked_array(data=[1, 2, 3],
             mask=False,
       fill_value=999999)
>>> x = np.array([(1, 1.), (2, 2.)], dtype=[('a',int), ('b', float)])
>>> x.view(ma.MaskedArray)
masked_array(data=[(1, 1.0), (2, 2.0)],
             mask=[(False, False), (False, False)],
       fill_value=(999999, 1.e+20),
            dtype=[('a', '<i8'), ('b', '<f8')])
```

- Yet another possibility is to use any of the following functions:

| | |
|---|---|
| *`asarray`*(a[, dtype, order]) | Convert the input to a masked array of the given data-type. |
| *`asanyarray`*(a[, dtype]) | Convert the input to a masked array, conserving subclasses. |
| *`fix_invalid`*(a[, mask, copy, fill_value]) | Return input with invalid data masked and replaced by a fill value. |
| *`masked_equal`*(x, value[, copy]) | Mask an array where equal to a given value. |
| *`masked_greater`*(x, value[, copy]) | Mask an array where greater than a given value. |
| *`masked_greater_equal`*(x, value[, copy]) | Mask an array where greater than or equal to a given value. |
| *`masked_inside`*(x, v1, v2[, copy]) | Mask an array inside a given interval. |
| *`masked_invalid`*(a[, copy]) | Mask an array where invalid values occur (NaNs or infs). |
| *`masked_less`*(x, value[, copy]) | Mask an array where less than a given value. |
| *`masked_less_equal`*(x, value[, copy]) | Mask an array where less than or equal to a given value. |
| *`masked_not_equal`*(x, value[, copy]) | Mask an array where *not* equal to a given value. |
| *`masked_object`*(x, value[, copy, shrink]) | Mask the array *x* where the data are exactly equal to value. |
| *`masked_outside`*(x, v1, v2[, copy]) | Mask an array outside a given interval. |
| *`masked_values`*(x, value[, rtol, atol, copy, …]) | Mask using floating point equality. |
| *`masked_where`*(condition, a[, copy]) | Mask an array where a condition is met. |

`numpy.ma.`**`asarray`**(*a*, *dtype=None*, *order=None*)

Convert the input to a masked array of the given data-type.

No copy is performed if the input is already an *ndarray*. If *a* is a subclass of *`MaskedArray`*, a base class *`MaskedArray`* is returned.

**Parameters**

**a**

[array_like] Input data, in any form that can be converted to a masked array. This includes lists, lists of tuples, tuples, tuples of tuples, tuples of lists, ndarrays and masked arrays.

**dtype**

[dtype, optional] By default, the data-type is inferred from the input data.

**order**

[{'C', 'F'}, optional] Whether to use row-major ('C') or column-major ('FORTRAN') memory representation. Default is 'C'.

**Returns**

**out**

[MaskedArray] Masked array interpretation of *a*.

**See also:**

***asanyarray***

Similar to *asarray*, but conserves subclasses.

## Examples

```
>>> x = np.arange(10.).reshape(2, 5)
>>> x
array([[0., 1., 2., 3., 4.],
       [5., 6., 7., 8., 9.]])
>>> np.ma.asarray(x)
masked_array(
  data=[[0., 1., 2., 3., 4.],
        [5., 6., 7., 8., 9.]],
  mask=False,
  fill_value=1e+20)
>>> type(np.ma.asarray(x))
<class 'numpy.ma.core.MaskedArray'>
```

`numpy.ma.`**`asanyarray`**(*a*, *dtype=None*)

Convert the input to a masked array, conserving subclasses.

If *a* is a subclass of *MaskedArray*, its class is conserved. No copy is performed if the input is already an *ndarray*.

**Parameters**

**a**

[array_like] Input data, in any form that can be converted to an array.

**dtype**

[dtype, optional] By default, the data-type is inferred from the input data.

**order**

[{'C', 'F'}, optional] Whether to use row-major ('C') or column-major ('FORTRAN') memory representation. Default is 'C'.

**Returns**

**out**

[MaskedArray] MaskedArray interpretation of *a*.

**See also:**

***asarray***

Similar to *asanyarray*, but does not conserve subclass.

## Examples

```
>>> x = np.arange(10.).reshape(2, 5)
>>> x
array([[0., 1., 2., 3., 4.],
       [5., 6., 7., 8., 9.]])
>>> np.ma.asanyarray(x)
masked_array(
  data=[[0., 1., 2., 3., 4.],
        [5., 6., 7., 8., 9.]],
  mask=False,
  fill_value=1e+20)
>>> type(np.ma.asanyarray(x))
<class 'numpy.ma.core.MaskedArray'>
```

`numpy.ma.`**`fix_invalid`**`(`*a*, *mask=False*, *copy=True*, *fill_value=None*`)`

Return input with invalid data masked and replaced by a fill value.

Invalid data means values of *nan*, *inf*, etc.

**Parameters**

**a**

[array_like] Input array, a (subclass of) ndarray.

**mask**

[sequence, optional] Mask. Must be convertible to an array of booleans with the same shape as *data*. True indicates a masked (i.e. invalid) data.

**copy**

[bool, optional] Whether to use a copy of *a* (True) or to fix *a* in place (False). Default is True.

**fill_value**

[scalar, optional] Value used for fixing invalid data. Default is None, in which case the `a.fill_value` is used.

**Returns**

**b**

[MaskedArray] The input array with invalid entries fixed.

### Notes

A copy is performed by default.

### Examples

```
>>> x = np.ma.array([1., -1, np.nan, np.inf], mask=[1] + [0]*3)
>>> x
masked_array(data=[--, -1.0, nan, inf],
             mask=[ True, False, False, False],
       fill_value=1e+20)
>>> np.ma.fix_invalid(x)
masked_array(data=[--, -1.0, --, --],
             mask=[ True, False,  True,  True],
       fill_value=1e+20)
```

```
>>> fixed = np.ma.fix_invalid(x)
>>> fixed.data
array([ 1.e+00, -1.e+00,  1.e+20,  1.e+20])
>>> x.data
array([ 1., -1., nan, inf])
```

`numpy.ma.`**`masked_equal`**(*x*, *value*, *copy=True*)

Mask an array where equal to a given value.

This function is a shortcut to `masked_where`, with *condition* = (x == value). For floating point arrays, consider using `masked_values(x, value)`.

**See also:**

***masked_where***

Mask where a condition is met.

***masked_values***

Mask using floating point equality.

### Examples

```
>>> import numpy.ma as ma
>>> a = np.arange(4)
>>> a
array([0, 1, 2, 3])
>>> ma.masked_equal(a, 2)
masked_array(data=[0, 1, --, 3],
             mask=[False, False,  True, False],
       fill_value=2)
```

`numpy.ma.`**`masked_greater`**(*x*, *value*, *copy=True*)

Mask an array where greater than a given value.

This function is a shortcut to `masked_where`, with *condition* = (x > value).

**See also:**

***masked_where***

Mask where a condition is met.

### Examples

```
>>> import numpy.ma as ma
>>> a = np.arange(4)
>>> a
array([0, 1, 2, 3])
>>> ma.masked_greater(a, 2)
masked_array(data=[0, 1, 2, --],
             mask=[False, False, False,  True],
       fill_value=999999)
```

`numpy.ma.`**`masked_greater_equal`**(*x*, *value*, *copy=True*)

Mask an array where greater than or equal to a given value.

This function is a shortcut to `masked_where`, with *condition* = (x >= value).

**See also:**

***masked_where***

Mask where a condition is met.

### Examples

```
>>> import numpy.ma as ma
>>> a = np.arange(4)
>>> a
array([0, 1, 2, 3])
>>> ma.masked_greater_equal(a, 2)
masked_array(data=[0, 1, --, --],
             mask=[False, False,  True,  True],
       fill_value=999999)
```

`numpy.ma.`**`masked_inside`**(*x*, *v1*, *v2*, *copy=True*)

Mask an array inside a given interval.

Shortcut to `masked_where`, where *condition* is True for *x* inside the interval [v1,v2] (v1 <= x <= v2). The boundaries *v1* and *v2* can be given in either order.

**See also:**

***masked_where***

Mask where a condition is met.

### Notes

The array *x* is prefilled with its filling value.

### Examples

```
>>> import numpy.ma as ma
>>> x = [0.31, 1.2, 0.01, 0.2, -0.4, -1.1]
>>> ma.masked_inside(x, -0.3, 0.3)
masked_array(data=[0.31, 1.2, --, --, -0.4, -1.1],
             mask=[False, False,  True,  True, False, False],
       fill_value=1e+20)
```

The order of *v1* and *v2* doesn't matter.

```
>>> ma.masked_inside(x, 0.3, -0.3)
masked_array(data=[0.31, 1.2, --, --, -0.4, -1.1],
             mask=[False, False,  True,  True, False, False],
       fill_value=1e+20)
```

`numpy.ma.`**`masked_invalid`**(*a*, *copy=True*)

Mask an array where invalid values occur (NaNs or infs).

This function is a shortcut to `masked_where`, with *condition* = ~(np.isfinite(a)). Any pre-existing mask is conserved. Only applies to arrays with a dtype where NaNs or infs make sense (i.e. floating point types), but accepts any array_like object.

**See also:**

***`masked_where`***

Mask where a condition is met.

### Examples

```
>>> import numpy.ma as ma
>>> a = np.arange(5, dtype=float)
>>> a[2] = np.NaN
>>> a[3] = np.PINF
>>> a
array([ 0.,   1., nan, inf,   4.])
>>> ma.masked_invalid(a)
masked_array(data=[0.0, 1.0, --, --, 4.0],
             mask=[False, False,  True,  True, False],
       fill_value=1e+20)
```

`numpy.ma.`**`masked_less`**(*x*, *value*, *copy=True*)

Mask an array where less than a given value.

This function is a shortcut to `masked_where`, with *condition* = (x < value).

**See also:**

***`masked_where`***

Mask where a condition is met.

### Examples

```
>>> import numpy.ma as ma
>>> a = np.arange(4)
>>> a
array([0, 1, 2, 3])
>>> ma.masked_less(a, 2)
masked_array(data=[--, --, 2, 3],
             mask=[ True,  True, False, False],
       fill_value=999999)
```

`numpy.ma.`**`masked_less_equal`**(*x*, *value*, *copy=True*)

Mask an array where less than or equal to a given value.

This function is a shortcut to `masked_where`, with *condition* = (x <= value).

**See also:**

***`masked_where`***

Mask where a condition is met.

### Examples

```
>>> import numpy.ma as ma
>>> a = np.arange(4)
>>> a
array([0, 1, 2, 3])
>>> ma.masked_less_equal(a, 2)
masked_array(data=[--, --, --, 3],
             mask=[ True,  True,  True, False],
       fill_value=999999)
```

`numpy.ma.`**`masked_not_equal`**(*x*, *value*, *copy=True*)

Mask an array where *not* equal to a given value.

This function is a shortcut to `masked_where`, with *condition* = (x != value).

**See also:**

***`masked_where`***

Mask where a condition is met.

### Examples

```
>>> import numpy.ma as ma
>>> a = np.arange(4)
>>> a
array([0, 1, 2, 3])
>>> ma.masked_not_equal(a, 2)
masked_array(data=[--, --, 2, --],
             mask=[ True,  True, False,  True],
       fill_value=999999)
```

`numpy.ma.`**`masked_object`**(*x*, *value*, *copy=True*, *shrink=True*)

Mask the array *x* where the data are exactly equal to value.

This function is similar to `masked_values`, but only suitable for object arrays: for floating point, use `masked_values` instead.

**Parameters**

**x**

[array_like] Array to mask

**value**

[object] Comparison value

**copy**

[{True, False}, optional] Whether to return a copy of *x*.

**shrink**

[{True, False}, optional] Whether to collapse a mask full of False to nomask

**Returns**

**result**

[MaskedArray] The result of masking *x* where equal to *value*.

**See also:**

**`masked_where`**

Mask where a condition is met.

**`masked_equal`**

Mask where equal to a given value (integers).

**`masked_values`**

Mask using floating point equality.

## Examples

```
>>> import numpy.ma as ma
>>> food = np.array(['green_eggs', 'ham'], dtype=object)
>>> # don't eat spoiled food
>>> eat = ma.masked_object(food, 'green_eggs')
>>> eat
masked_array(data=[--, 'ham'],
             mask=[ True, False],
       fill_value='green_eggs',
            dtype=object)
>>> # plain ol` ham is boring
>>> fresh_food = np.array(['cheese', 'ham', 'pineapple'], dtype=object)
>>> eat = ma.masked_object(fresh_food, 'green_eggs')
>>> eat
masked_array(data=['cheese', 'ham', 'pineapple'],
             mask=False,
```

(continues on next page)

(continued from previous page)

```
       fill_value='green_eggs',
            dtype=object)
```

Note that *mask* is set to `nomask` if possible.

```
>>> eat
masked_array(data=['cheese', 'ham', 'pineapple'],
             mask=False,
       fill_value='green_eggs',
            dtype=object)
```

`numpy.ma.`**`masked_outside`**(*x*, *v1*, *v2*, *copy=True*)

Mask an array outside a given interval.

Shortcut to `masked_where`, where *condition* is True for *x* outside the interval [v1,v2] (x < v1)|(x > v2). The boundaries *v1* and *v2* can be given in either order.

**See also:**

***`masked_where`***

Mask where a condition is met.

### Notes

The array *x* is prefilled with its filling value.

### Examples

```
>>> import numpy.ma as ma
>>> x = [0.31, 1.2, 0.01, 0.2, -0.4, -1.1]
>>> ma.masked_outside(x, -0.3, 0.3)
masked_array(data=[--, --, 0.01, 0.2, --, --],
             mask=[ True,  True, False, False,  True,  True],
       fill_value=1e+20)
```

The order of *v1* and *v2* doesn't matter.

```
>>> ma.masked_outside(x, 0.3, -0.3)
masked_array(data=[--, --, 0.01, 0.2, --, --],
             mask=[ True,  True, False, False,  True,  True],
       fill_value=1e+20)
```

`numpy.ma.`**`masked_values`**(*x*, *value*, *rtol=1e-05*, *atol=1e-08*, *copy=True*, *shrink=True*)

Mask using floating point equality.

Return a MaskedArray, masked where the data in array *x* are approximately equal to *value*, determined using *isclose*. The default tolerances for *`masked_values`* are the same as those for *isclose*.

For integer types, exact equality is used, in the same way as *`masked_equal`*.

The fill_value is set to *value* and the mask is set to `nomask` if possible.

**Parameters**

**x**

[array_like] Array to mask.

**value**

[float] Masking value.

**rtol, atol**

[float, optional] Tolerance parameters passed on to *isclose*

**copy**

[bool, optional] Whether to return a copy of *x*.

**shrink**

[bool, optional] Whether to collapse a mask full of False to `nomask`.

**Returns**

**result**

[MaskedArray] The result of masking *x* where approximately equal to *value*.

**See also:**

***masked_where***

Mask where a condition is met.

***masked_equal***

Mask where equal to a given value (integers).

## Examples

```
>>> import numpy.ma as ma
>>> x = np.array([1, 1.1, 2, 1.1, 3])
>>> ma.masked_values(x, 1.1)
masked_array(data=[1.0, --, 2.0, --, 3.0],
             mask=[False,  True, False,  True, False],
       fill_value=1.1)
```

Note that *mask* is set to `nomask` if possible.

```
>>> ma.masked_values(x, 1.5)
masked_array(data=[1. , 1.1, 2. , 1.1, 3. ],
             mask=False,
       fill_value=1.5)
```

For integers, the fill value will be different in general to the result of `masked_equal`.

```
>>> x = np.arange(5)
>>> x
array([0, 1, 2, 3, 4])
>>> ma.masked_values(x, 2)
masked_array(data=[0, 1, --, 3, 4],
             mask=[False, False,  True, False, False],
```

(continues on next page)

(continued from previous page)

```
       fill_value=2)
>>> ma.masked_equal(x, 2)
masked_array(data=[0, 1, --, 3, 4],
             mask=[False, False,  True, False, False],
       fill_value=2)
```

`numpy.ma.`**`masked_where`**(*condition*, *a*, *copy=True*)

Mask an array where a condition is met.

Return *a* as an array masked where *condition* is True. Any masked values of *a* or *condition* are also masked in the output.

**Parameters**

**condition**

[array_like] Masking condition. When *condition* tests floating point values for equality, consider using `masked_values` instead.

**a**

[array_like] Array to mask.

**copy**

[bool] If True (default) make a copy of *a* in the result. If False modify *a* in place and return a view.

**Returns**

**result**

[MaskedArray] The result of masking *a* where *condition* is True.

**See also:**

***`masked_values`***

Mask using floating point equality.

***`masked_equal`***

Mask where equal to a given value.

***`masked_not_equal`***

Mask where *not* equal to a given value.

***`masked_less_equal`***

Mask where less than or equal to a given value.

***`masked_greater_equal`***

Mask where greater than or equal to a given value.

***`masked_less`***

Mask where less than a given value.

***`masked_greater`***

Mask where greater than a given value.

***masked_inside***

Mask inside a given interval.

***masked_outside***

Mask outside a given interval.

***masked_invalid***

Mask invalid values (NaNs or infs).

### Examples

```
>>> import numpy.ma as ma
>>> a = np.arange(4)
>>> a
array([0, 1, 2, 3])
>>> ma.masked_where(a <= 2, a)
masked_array(data=[--, --, --, 3],
             mask=[ True,  True,  True, False],
       fill_value=999999)
```

Mask array *b* conditional on *a*.

```
>>> b = ['a', 'b', 'c', 'd']
>>> ma.masked_where(a == 2, b)
masked_array(data=['a', 'b', --, 'd'],
             mask=[False, False,  True, False],
       fill_value='N/A',
            dtype='<U1')
```

Effect of the `copy` argument.

```
>>> c = ma.masked_where(a <= 2, a)
>>> c
masked_array(data=[--, --, --, 3],
             mask=[ True,  True,  True, False],
       fill_value=999999)
>>> c[0] = 99
>>> c
masked_array(data=[99, --, --, 3],
             mask=[False,  True,  True, False],
       fill_value=999999)
>>> a
array([0, 1, 2, 3])
>>> c = ma.masked_where(a <= 2, a, copy=False)
>>> c[0] = 99
>>> c
masked_array(data=[99, --, --, 3],
             mask=[False,  True,  True, False],
       fill_value=999999)
>>> a
array([99,  1,  2,  3])
```

When *condition* or *a* contain masked values.

```
>>> a = np.arange(4)
>>> a = ma.masked_where(a == 2, a)
>>> a
masked_array(data=[0, 1, --, 3],
             mask=[False, False,  True, False],
       fill_value=999999)
>>> b = np.arange(4)
>>> b = ma.masked_where(b == 0, b)
>>> b
masked_array(data=[--, 1, 2, 3],
             mask=[ True, False, False, False],
       fill_value=999999)
>>> ma.masked_where(a == 3, b)
masked_array(data=[--, 1, --, --],
             mask=[ True, False,  True,  True],
       fill_value=999999)
```

### Accessing the data

The underlying data of a masked array can be accessed in several ways:

- through the *`data`* attribute. The output is a view of the array as a *`numpy.ndarray`* or one of its subclasses, depending on the type of the underlying data at the masked array creation.
- through the *`__array__`* method. The output is then a *`numpy.ndarray`*.
- by directly taking a view of the masked array as a *`numpy.ndarray`* or one of its subclass (which is actually what using the *`data`* attribute does).
- by using the *`getdata`* function.

None of these methods is completely satisfactory if some entries have been marked as invalid. As a general rule, where a representation of the array is required without any masked entries, it is recommended to fill the array with the *`filled`* method.

### Accessing the mask

The mask of a masked array is accessible through its *`mask`* attribute. We must keep in mind that a `True` entry in the mask indicates an *invalid* data.

Another possibility is to use the *`getmask`* and *`getmaskarray`* functions. `getmask(x)` outputs the mask of `x` if `x` is a masked array, and the special value *`nomask`* otherwise. `getmaskarray(x)` outputs the mask of `x` if `x` is a masked array. If `x` has no invalid entry or is not a masked array, the function outputs a boolean array of `False` with as many elements as `x`.

### Accessing only the valid entries

To retrieve only the valid entries, we can use the inverse of the mask as an index. The inverse of the mask can be calculated with the *`numpy.logical_not`* function or simply with the ~ operator:

```
>>> x = ma.array([[1, 2], [3, 4]], mask=[[0, 1], [1, 0]])
>>> x[~x.mask]
masked_array(data=[1, 4],
             mask=[False, False],
       fill_value=999999)
```

Another way to retrieve the valid data is to use the `compressed` method, which returns a one-dimensional `ndarray` (or one of its subclasses, depending on the value of the `baseclass` attribute):

```
>>> x.compressed()
array([1, 4])
```

Note that the output of `compressed` is always 1D.

## Modifying the mask

### Masking an entry

The recommended way to mark one or several specific entries of a masked array as invalid is to assign the special value `masked` to them:

```
>>> x = ma.array([1, 2, 3])
>>> x[0] = ma.masked
>>> x
masked_array(data=[--, 2, 3],
             mask=[ True, False, False],
       fill_value=999999)
>>> y = ma.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
>>> y[(0, 1, 2), (1, 2, 0)] = ma.masked
>>> y
masked_array(
  data=[[1, --, 3],
        [4, 5, --],
        [--, 8, 9]],
  mask=[[False,  True, False],
        [False, False,  True],
        [ True, False, False]],
  fill_value=999999)
>>> z = ma.array([1, 2, 3, 4])
>>> z[:-2] = ma.masked
>>> z
masked_array(data=[--, --, 3, 4],
             mask=[ True,  True, False, False],
       fill_value=999999)
```

A second possibility is to modify the `mask` directly, but this usage is discouraged.

---

**Note:** When creating a new masked array with a simple, non-structured datatype, the mask is initially set to the special value `nomask`, that corresponds roughly to the boolean `False`. Trying to set an element of `nomask` will fail with a `TypeError` exception, as a boolean does not support item assignment.

---

All the entries of an array can be masked at once by assigning `True` to the mask:

```
>>> x = ma.array([1, 2, 3], mask=[0, 0, 1])
>>> x.mask = True
>>> x
masked_array(data=[--, --, --],
             mask=[ True,  True,  True],
       fill_value=999999,
            dtype=int64)
```

Finally, specific entries can be masked and/or unmasked by assigning to the mask a sequence of booleans:

```
>>> x = ma.array([1, 2, 3])
>>> x.mask = [0, 1, 0]
>>> x
masked_array(data=[1, --, 3],
             mask=[False,  True, False],
       fill_value=999999)
```

### Unmasking an entry

To unmask one or several specific entries, we can just assign one or several new valid values to them:

```
>>> x = ma.array([1, 2, 3], mask=[0, 0, 1])
>>> x
masked_array(data=[1, 2, --],
             mask=[False, False,  True],
       fill_value=999999)
>>> x[-1] = 5
>>> x
masked_array(data=[1, 2, 5],
             mask=[False, False, False],
       fill_value=999999)
```

---

**Note:** Unmasking an entry by direct assignment will silently fail if the masked array has a *hard* mask, as shown by the `hardmask` attribute. This feature was introduced to prevent overwriting the mask. To force the unmasking of an entry where the array has a hard mask, the mask must first to be softened using the *`soften_mask`* method before the allocation. It can be re-hardened with *`harden_mask`*:

```
>>> x = ma.array([1, 2, 3], mask=[0, 0, 1], hard_mask=True)
>>> x
masked_array(data=[1, 2, --],
             mask=[False, False,  True],
       fill_value=999999)
>>> x[-1] = 5
>>> x
masked_array(data=[1, 2, --],
             mask=[False, False,  True],
       fill_value=999999)
>>> x.soften_mask()
masked_array(data=[1, 2, --],
             mask=[False, False,  True],
       fill_value=999999)
>>> x[-1] = 5
>>> x
masked_array(data=[1, 2, 5],
             mask=[False, False, False],
       fill_value=999999)
>>> x.harden_mask()
masked_array(data=[1, 2, 5],
             mask=[False, False, False],
       fill_value=999999)
```

---

To unmask all masked entries of a masked array (provided the mask isn't a hard mask), the simplest solution is to assign the constant *`nomask`* to the mask:

```
>>> x = ma.array([1, 2, 3], mask=[0, 0, 1])
>>> x
masked_array(data=[1, 2, --],
             mask=[False, False,  True],
       fill_value=999999)
>>> x.mask = ma.nomask
>>> x
masked_array(data=[1, 2, 3],
             mask=[False, False, False],
       fill_value=999999)
```

### Indexing and slicing

As a *`MaskedArray`* is a subclass of *`numpy.ndarray`*, it inherits its mechanisms for indexing and slicing.

When accessing a single entry of a masked array with no named fields, the output is either a scalar (if the corresponding entry of the mask is `False`) or the special value *`masked`* (if the corresponding entry of the mask is `True`):

```
>>> x = ma.array([1, 2, 3], mask=[0, 0, 1])
>>> x[0]
1
>>> x[-1]
masked
>>> x[-1] is ma.masked
True
```

If the masked array has named fields, accessing a single entry returns a `numpy.void` object if none of the fields are masked, or a 0d masked array with the same dtype as the initial array if at least one of the fields is masked.

```
>>> y = ma.masked_array([(1,2), (3, 4)],
...                mask=[(0, 0), (0, 1)],
...               dtype=[('a', int), ('b', int)])
>>> y[0]
(1, 2)
>>> y[-1]
(3, --)
```

When accessing a slice, the output is a masked array whose *`data`* attribute is a view of the original data, and whose mask is either *`nomask`* (if there was no invalid entries in the original array) or a view of the corresponding slice of the original mask. The view is required to ensure propagation of any modification of the mask to the original.

```
>>> x = ma.array([1, 2, 3, 4, 5], mask=[0, 1, 0, 0, 1])
>>> mx = x[:3]
>>> mx
masked_array(data=[1, --, 3],
             mask=[False,  True, False],
       fill_value=999999)
>>> mx[1] = -1
>>> mx
masked_array(data=[1, -1, 3],
             mask=[False, False, False],
       fill_value=999999)
>>> x.mask
array([False, False, False, False,  True])
>>> x.data
array([ 1, -1,  3,  4,  5])
```

Accessing a field of a masked array with structured datatype returns a `MaskedArray`.

### Operations on masked arrays

Arithmetic and comparison operations are supported by masked arrays. As much as possible, invalid entries of a masked array are not processed, meaning that the corresponding `data` entries *should* be the same before and after the operation.

> **Warning:** We need to stress that this behavior may not be systematic, that masked data may be affected by the operation in some cases and therefore users should not rely on this data remaining unchanged.

The `numpy.ma` module comes with a specific implementation of most ufuncs. Unary and binary functions that have a validity domain (such as `log` or `divide`) return the `masked` constant whenever the input is masked or falls outside the validity domain:

```
>>> ma.log([-1, 0, 1, 2])
masked_array(data=[--, --, 0.0, 0.6931471805599453],
             mask=[ True,  True, False, False],
       fill_value=1e+20)
```

Masked arrays also support standard numpy ufuncs. The output is then a masked array. The result of a unary ufunc is masked wherever the input is masked. The result of a binary ufunc is masked wherever any of the input is masked. If the ufunc also returns the optional context output (a 3-element tuple containing the name of the ufunc, its arguments and its domain), the context is processed and entries of the output masked array are masked wherever the corresponding input fall outside the validity domain:

```
>>> x = ma.array([-1, 1, 0, 2, 3], mask=[0, 0, 0, 0, 1])
>>> np.log(x)
masked_array(data=[--, 0.0, --, 0.6931471805599453, --],
             mask=[ True, False,  True, False,  True],
       fill_value=1e+20)
```

## 1.7.3 Examples

### Data with a given value representing missing data

Let's consider a list of elements, `x`, where values of -9999. represent missing data. We wish to compute the average value of the data and the vector of anomalies (deviations from the average):

```
>>> import numpy.ma as ma
>>> x = [0.,1.,-9999.,3.,4.]
>>> mx = ma.masked_values (x, -9999.)
>>> print(mx.mean())
2.0
>>> print(mx - mx.mean())
[-2.0 -1.0 -- 1.0 2.0]
>>> print(mx.anom())
[-2.0 -1.0 -- 1.0 2.0]
```

### Filling in the missing data

Suppose now that we wish to print that same data, but with the missing values replaced by the average value.

```
>>> print(mx.filled(mx.mean()))
[ 0.  1.  2.  3.  4.]
```

### Numerical operations

Numerical operations can be easily performed without worrying about missing values, dividing by zero, square roots of negative numbers, etc.:

```
>>> import numpy.ma as ma
>>> x = ma.array([1., -1., 3., 4., 5., 6.], mask=[0,0,0,0,1,0])
>>> y = ma.array([1., 2., 0., 4., 5., 6.], mask=[0,0,0,0,0,1])
>>> print(ma.sqrt(x/y))
[1.0 -- -- 1.0 -- --]
```

Four values of the output are invalid: the first one comes from taking the square root of a negative number, the second from the division by zero, and the last two where the inputs were masked.

### Ignoring extreme values

Let's consider an array `d` of floats between 0 and 1. We wish to compute the average of the values of `d` while ignoring any data outside the range `[0.2, 0.9]`:

```
>>> d = np.linspace(0, 1, 20)
>>> print(d.mean() - ma.masked_outside(d, 0.2, 0.9).mean())
-0.05263157894736836
```

## 1.7.4 Constants of the `numpy.ma` module

In addition to the *`MaskedArray`* class, the *`numpy.ma`* module defines several constants.

`numpy.ma.`**`masked`**

The *`masked`* constant is a special case of *`MaskedArray`*, with a float datatype and a null shape. It is used to test whether a specific entry of a masked array is masked, or to mask one or several entries of a masked array:

```
>>> x = ma.array([1, 2, 3], mask=[0, 1, 0])
>>> x[1] is ma.masked
True
>>> x[-1] = ma.masked
>>> x
masked_array(data=[1, --, --],
             mask=[False,  True,  True],
       fill_value=999999)
```

`numpy.ma.`**`nomask`**

Value indicating that a masked array has no invalid entry. *`nomask`* is used internally to speed up computations when the mask is not needed. It is represented internally as `np.False_`.

`numpy.ma.`**`masked_print_options`**

String used in lieu of missing data when a masked array is printed. By default, this string is `'--'`.

## 1.7.5 The `MaskedArray` class

**class** numpy.ma.**MaskedArray**

A subclass of *ndarray* designed to manipulate numerical arrays with missing data.

An instance of *MaskedArray* can be thought as the combination of several elements:

- The *data*, as a regular *numpy.ndarray* of any shape or datatype (the data).
- A boolean *mask* with the same shape as the data, where a `True` value indicates that the corresponding element of the data is invalid. The special value *nomask* is also acceptable for arrays without named fields, and indicates that no data is invalid.
- A *fill_value*, a value that may be used to replace the invalid entries in order to return a standard *numpy.ndarray*.

### Attributes and properties of masked arrays

**See also:**

*Array Attributes*

MaskedArray.**data**

Returns the underlying data, as a view of the masked array.

If the underlying data is a subclass of *numpy.ndarray*, it is returned as such.

```
>>> x = np.ma.array(np.matrix([[1, 2], [3, 4]]), mask=[[0, 1], [1, 0]])
>>> x.data
matrix([[1, 2],
        [3, 4]])
```

The type of the data can be accessed through the *baseclass* attribute.

MaskedArray.**mask**

Current mask.

MaskedArray.**recordmask**

Get or set the mask of the array if it has no named fields. For structured arrays, returns a ndarray of booleans where entries are `True` if **all** the fields are masked, `False` otherwise:

```
>>> x = np.ma.array([(1, 1), (2, 2), (3, 3), (4, 4), (5, 5)],
...         mask=[(0, 0), (1, 0), (1, 1), (0, 1), (0, 0)],
...        dtype=[('a', int), ('b', int)])
>>> x.recordmask
array([False, False,  True, False, False])
```

MaskedArray.**fill_value**

The filling value of the masked array is a scalar. When setting, None will set to a default based on the data type.

### Examples

```
>>> for dt in [np.int32, np.int64, np.float64, np.complex128]:
...     np.ma.array([0, 1], dtype=dt).get_fill_value()
...
999999
999999
1e+20
(1e+20+0j)
```

```
>>> x = np.ma.array([0, 1.], fill_value=-np.inf)
>>> x.fill_value
-inf
>>> x.fill_value = np.pi
>>> x.fill_value
3.1415926535897931 # may vary
```

Reset to default:

```
>>> x.fill_value = None
>>> x.fill_value
1e+20
```

`MaskedArray.`**`baseclass`**

Class of the underlying data (read-only).

`MaskedArray.`**`sharedmask`**

Share status of the mask (read-only).

`MaskedArray.`**`hardmask`**

Hardness of the mask

As *`MaskedArray`* is a subclass of *`ndarray`*, a masked array also inherits all the attributes and properties of a *`ndarray`* instance.

| | |
|---|---|
| *`MaskedArray.base`* | Base object if memory is from some other object. |
| *`MaskedArray.ctypes`* | An object to simplify the interaction of the array with the ctypes module. |
| *`MaskedArray.dtype`* | Data-type of the array's elements. |
| *`MaskedArray.flags`* | Information about the memory layout of the array. |
| *`MaskedArray.itemsize`* | Length of one array element in bytes. |
| *`MaskedArray.nbytes`* | Total bytes consumed by the elements of the array. |
| *`MaskedArray.ndim`* | Number of array dimensions. |
| *`MaskedArray.shape`* | Tuple of array dimensions. |
| *`MaskedArray.size`* | Number of elements in the array. |
| *`MaskedArray.strides`* | Tuple of bytes to step in each dimension when traversing an array. |
| *`MaskedArray.imag`* | The imaginary part of the masked array. |
| *`MaskedArray.real`* | The real part of the masked array. |
| *`MaskedArray.flat`* | Return a flat iterator, or set a flattened version of self to value. |
| *`MaskedArray.__array_priority__`* | |

attribute

`MaskedArray.`**`base`**

Base object if memory is from some other object.

### Examples

The base of an array that owns its memory is None:

```
>>> x = np.array([1,2,3,4])
>>> x.base is None
True
```

Slicing creates a view, whose memory is shared with x:

```
>>> y = x[2:]
>>> y.base is x
True
```

attribute

`MaskedArray.`**`ctypes`**

An object to simplify the interaction of the array with the ctypes module.

This attribute creates an object that makes it easier to use arrays when calling shared libraries with the ctypes module. The returned object has, among others, data, shape, and strides attributes (see Notes below) which themselves return ctypes objects that can be used as arguments to a shared library.

**Parameters**

**None**

**Returns**

**c**

[Python object] Possessing attributes data, shape, strides, etc.

**See also:**

*`numpy.ctypeslib`*

### Notes

Below are the public attributes of this object which were documented in “Guide to NumPy” (we have omitted undocumented public attributes, as well as documented private attributes):

`_ctypes.`**`data`**

A pointer to the memory area of the array as a Python integer. This memory area may contain data that is not aligned, or not in correct byte-order. The memory area may not even be writeable. The array flags and data-type of this array should be respected when passing this attribute to arbitrary C-code to avoid trouble that can include Python crashing. User Beware! The value of this attribute is exactly the same as `self._array_interface_['data'][0]`.

Note that unlike `data_as`, a reference will not be kept to the array: code like `ctypes.c_void_p((a + b).ctypes.data)` will result in a pointer to a deallocated array, and should be spelt `(a + b).ctypes.data_as(ctypes.c_void_p)` `_ctypes.`**`shape`**

(c_intp*self.ndim): A ctypes array of length self.ndim where the basetype is the C-integer corresponding to `dtype('p')` on this platform. This base-type could be `ctypes.c_int`, `ctypes.c_long`, or `ctypes.c_longlong` depending on the platform. The c_intp type is defined accordingly in *`numpy.ctypeslib`*. The ctypes array contains the shape of the underlying array.

`_ctypes.`**`strides`**

(c_intp*self.ndim): A ctypes array of length self.ndim where the basetype is the same as for the shape attribute. This ctypes array contains the strides information from the underlying array. This strides information is important for showing how many bytes must be jumped to get to the next element in the array.

`_ctypes.`**`data_as`**(*self*, *obj*)

Return the data pointer cast to a particular c-types object. For example, calling `self._as_parameter_` is equivalent to `self.data_as(ctypes.c_void_p)`. Perhaps you want to use the data as a pointer to a ctypes array of floating-point data: `self.data_as(ctypes.POINTER(ctypes.c_double))`.

The returned pointer will keep a reference to the array.

`_ctypes.`**`shape_as`**(*self*, *obj*)

Return the shape tuple as an array of some other c-types type. For example: `self.shape_as(ctypes.c_short)`.

`_ctypes.`**`strides_as`**(*self*, *obj*)

Return the strides tuple as an array of some other c-types type. For example: `self.strides_as(ctypes.c_longlong)`.

If the ctypes module is not available, then the ctypes attribute of array objects still returns something useful, but ctypes objects are not returned and errors may be raised instead. In particular, the object will still have the `as_parameter` attribute which will return an integer equal to the data attribute.

### Examples

```
>>> import ctypes
>>> x = np.array([[0, 1], [2, 3]], dtype=np.int32)
>>> x
array([[0, 1],
       [2, 3]], dtype=int32)
>>> x.ctypes.data
31962608 # may vary
>>> x.ctypes.data_as(ctypes.POINTER(ctypes.c_uint32))
<__main__.LP_c_uint object at 0x7ff2fc1fc200> # may vary
>>> x.ctypes.data_as(ctypes.POINTER(ctypes.c_uint32)).contents
c_uint(0)
>>> x.ctypes.data_as(ctypes.POINTER(ctypes.c_uint64)).contents
c_ulong(4294967296)
>>> x.ctypes.shape
<numpy.core._internal.c_long_Array_2 object at 0x7ff2fc1fce60> # may vary
>>> x.ctypes.strides
<numpy.core._internal.c_long_Array_2 object at 0x7ff2fc1ff320> # may vary
```

property

**`property`** `MaskedArray.`**`dtype`**

Data-type of the array's elements.

**Parameters**

**None**

**Returns**

**d**

[numpy dtype object]

**See also:**

*numpy.dtype*

### Examples

```
>>> x
array([[0, 1],
       [2, 3]])
>>> x.dtype
dtype('int32')
>>> type(x.dtype)
<type 'numpy.dtype'>
```

attribute

`MaskedArray.`**`flags`**

Information about the memory layout of the array.

### Notes

The *flags* object can be accessed dictionary-like (as in `a.flags['WRITEABLE']`), or by using lowercased attribute names (as in `a.flags.writeable`). Short flag names are only supported in dictionary access.

Only the WRITEBACKIFCOPY, UPDATEIFCOPY, WRITEABLE, and ALIGNED flags can be changed by the user, via direct assignment to the attribute or dictionary entry, or by calling *ndarray.setflags*.

The array flags cannot be set arbitrarily:

- UPDATEIFCOPY can only be set `False`.
- WRITEBACKIFCOPY can only be set `False`.
- ALIGNED can only be set `True` if the data is truly aligned.
- WRITEABLE can only be set `True` if the array owns its own memory or the ultimate owner of the memory exposes a writeable buffer interface or is a string.

Arrays can be both C-style and Fortran-style contiguous simultaneously. This is clear for 1-dimensional arrays, but can also be true for higher dimensional arrays.

Even for contiguous arrays a stride for a given dimension `arr.strides[dim]` may be *arbitrary* if `arr.shape[dim] == 1` or the array has no elements. It does *not* generally hold that `self.strides[-1] == self.itemsize` for C-style contiguous arrays or `self.strides[0] == self.itemsize` for Fortran-style contiguous arrays is true.

**Attributes**

**C_CONTIGUOUS (C)**

The data is in a single, C-style contiguous segment.

**F_CONTIGUOUS (F)**

The data is in a single, Fortran-style contiguous segment.

**OWNDATA (O)**

The array owns the memory it uses or borrows it from another object.

**WRITEABLE (W)**

The data area can be written to. Setting this to False locks the data, making it read-only. A view (slice, etc.) inherits WRITEABLE from its base array at creation time, but a view of a writeable array may be subsequently locked while the base array remains writeable. (The opposite is not true, in that a view of a locked array may not be made writeable. However, currently, locking a base object does not lock any views that already reference it, so under that circumstance it is possible to alter the contents of a locked array via a previously created writeable view onto it.) Attempting to change a non-writeable array raises a RuntimeError exception.

**ALIGNED (A)**

The data and all elements are aligned appropriately for the hardware.

**WRITEBACKIFCOPY (X)**

This array is a copy of some other array. The C-API function PyArray_ResolveWritebackIfCopy must be called before deallocating to the base array will be updated with the contents of this array.

**UPDATEIFCOPY (U)**

(Deprecated, use WRITEBACKIFCOPY) This array is a copy of some other array. When this array is deallocated, the base array will be updated with the contents of this array.

**FNC**

F_CONTIGUOUS and not C_CONTIGUOUS.

**FORC**

F_CONTIGUOUS or C_CONTIGUOUS (one-segment test).

**BEHAVED (B)**

ALIGNED and WRITEABLE.

**CARRAY (CA)**

BEHAVED and C_CONTIGUOUS.

**FARRAY (FA)**

BEHAVED and F_CONTIGUOUS and not C_CONTIGUOUS.

attribute

`MaskedArray.`**`itemsize`**

Length of one array element in bytes.

### Examples

```
>>> x = np.array([1,2,3], dtype=np.float64)
>>> x.itemsize
8
>>> x = np.array([1,2,3], dtype=np.complex128)
>>> x.itemsize
16
```

attribute

`MaskedArray.`**`nbytes`**

Total bytes consumed by the elements of the array.

### Notes

Does not include memory consumed by non-element attributes of the array object.

### Examples

```
>>> x = np.zeros((3,5,2), dtype=np.complex128)
>>> x.nbytes
480
>>> np.prod(x.shape) * x.itemsize
480
```

attribute

`MaskedArray.`**`ndim`**

Number of array dimensions.

### Examples

```
>>> x = np.array([1, 2, 3])
>>> x.ndim
1
>>> y = np.zeros((2, 3, 4))
>>> y.ndim
3
```

property

**`property`** `MaskedArray.`**`shape`**

Tuple of array dimensions.

The shape property is usually used to get the current shape of an array, but may also be used to reshape the array in-place by assigning a tuple of array dimensions to it. As with `numpy.reshape`, one of the new shape dimensions can be -1, in which case its value is inferred from the size of the array and the remaining dimensions. Reshaping an array in-place will fail if a copy is required.

**See also:**

***`numpy.reshape`***

similar function

**ndarray.reshape**
similar method

### Examples

```
>>> x = np.array([1, 2, 3, 4])
>>> x.shape
(4,)
>>> y = np.zeros((2, 3, 4))
>>> y.shape
(2, 3, 4)
>>> y.shape = (3, 8)
>>> y
array([[ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.]])
>>> y.shape = (3, 6)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: total size of new array must be unchanged
>>> np.zeros((4,2))[::2].shape = (-1,)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: Incompatible shape for in-place modification. Use
`.reshape()` to make a copy with the desired shape.
```

attribute

`MaskedArray.`**size**

Number of elements in the array.

Equal to `np.prod(a.shape)`, i.e., the product of the array's dimensions.

### Notes

*a.size* returns a standard arbitrary precision Python integer. This may not be the case with other methods of obtaining the same value (like the suggested `np.prod(a.shape)`, which returns an instance of `np.int_`), and may be relevant if the value is used further in calculations that may overflow a fixed size integer type.

### Examples

```
>>> x = np.zeros((3, 5, 2), dtype=np.complex128)
>>> x.size
30
>>> np.prod(x.shape)
30
```

attribute

`MaskedArray.`**strides**

Tuple of bytes to step in each dimension when traversing an array.

The byte offset of element `(i[0], i[1], ..., i[n])` in an array *a* is:

```
offset = sum(np.array(i) * a.strides)
```

A more detailed explanation of strides can be found in the "ndarray.rst" file in the NumPy reference guide.

**See also:**

*numpy.lib.stride_tricks.as_strided*

### Notes

Imagine an array of 32-bit integers (each 4 bytes):

```
x = np.array([[0, 1, 2, 3, 4],
              [5, 6, 7, 8, 9]], dtype=np.int32)
```

This array is stored in memory as 40 bytes, one after the other (known as a contiguous block of memory). The strides of an array tell us how many bytes we have to skip in memory to move to the next position along a certain axis. For example, we have to skip 4 bytes (1 value) to move to the next column, but 20 bytes (5 values) to get to the same position in the next row. As such, the strides for the array *x* will be `(20, 4)`.

### Examples

```
>>> y = np.reshape(np.arange(2*3*4), (2,3,4))
>>> y
array([[[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [ 8,  9, 10, 11]],
       [[12, 13, 14, 15],
        [16, 17, 18, 19],
        [20, 21, 22, 23]]])
>>> y.strides
(48, 16, 4)
>>> y[1,1,1]
17
>>> offset=sum(y.strides * np.array((1,1,1)))
>>> offset/y.itemsize
17
```

```
>>> x = np.reshape(np.arange(5*6*7*8), (5,6,7,8)).transpose(2,3,1,0)
>>> x.strides
(32, 4, 224, 1344)
>>> i = np.array([3,5,2,2])
>>> offset = sum(i * x.strides)
>>> x[3,5,2,2]
813
>>> offset / x.itemsize
813
```

property

**property** `MaskedArray`**.imag**

The imaginary part of the masked array.

This property is a view on the imaginary part of this *MaskedArray*.

**See also:**

*real*

### Examples

```
>>> x = np.ma.array([1+1.j, -2j, 3.45+1.6j], mask=[False, True, False])
>>> x.imag
masked_array(data=[1.0, --, 1.6],
             mask=[False,  True, False],
       fill_value=1e+20)
```

property

**property** `MaskedArray.`**`real`**

The real part of the masked array.

This property is a view on the real part of this *MaskedArray*.

**See also:**

*imag*

### Examples

```
>>> x = np.ma.array([1+1.j, -2j, 3.45+1.6j], mask=[False, True, False])
>>> x.real
masked_array(data=[1.0, --, 3.45],
             mask=[False,  True, False],
       fill_value=1e+20)
```

property

**property** `MaskedArray.`**`flat`**

Return a flat iterator, or set a flattened version of self to value.

attribute

`MaskedArray.`**`__array_priority__ = 15`**

## 1.7.6 MaskedArray methods

**See also:**

*Array methods*

### Conversion

| | |
|---|---|
| *MaskedArray.__float__*(self) | Convert to float. |
| *MaskedArray.__int__*(self) | Convert to int. |
| *MaskedArray.view*(self[, dtype, type, fill_value]) | Return a view of the MaskedArray data. |
| *MaskedArray.astype*(dtype[, order, casting, …]) | Copy of the array, cast to a specified type. |
| *MaskedArray.byteswap*([inplace]) | Swap the bytes of the array elements |
| *MaskedArray.compressed*(self) | Return all the non-masked data as a 1-D array. |

Continued on next page

Table 53 – continued from previous page

| | |
|---|---|
| `MaskedArray.filled`(self[, fill_value]) | Return a copy of self, with masked values filled with a given value. |
| `MaskedArray.tofile`(self, fid[, sep, format]) | Save a masked array to a file in binary format. |
| `MaskedArray.toflex`(self) | Transforms a masked array into a flexible-type array. |
| `MaskedArray.tolist`(self[, fill_value]) | Return the data portion of the masked array as a hierarchical Python list. |
| `MaskedArray.torecords`(self) | Transforms a masked array into a flexible-type array. |
| `MaskedArray.tostring`(self[, fill_value, order]) | A compatibility alias for *tobytes*, with exactly the same behavior. |
| `MaskedArray.tobytes`(self[, fill_value, order]) | Return the array data as a string containing the raw bytes in the array. |

method

`MaskedArray.`**`__float__`**(*self*)

Convert to float.

method

`MaskedArray.`**`__int__`**(*self*)

Convert to int.

method

`MaskedArray.`**`view`**(*self*, *dtype=None*, *type=None*, *fill_value=None*)

Return a view of the MaskedArray data.

**Parameters**

**dtype**

[data-type or ndarray sub-class, optional] Data-type descriptor of the returned view, e.g., float32 or int16. The default, None, results in the view having the same data-type as *a*. As with `ndarray.view`, dtype can also be specified as an ndarray sub-class, which then specifies the type of the returned object (this is equivalent to setting the `type` parameter).

**type**

[Python type, optional] Type of the returned view, either ndarray or a subclass. The default None results in type preservation.

**fill_value**

[scalar, optional] The value to use for invalid entries (None by default). If None, then this argument is inferred from the passed `dtype`, or in its absence the original array, as discussed in the notes below.

**See also:**

**`numpy.ndarray.view`**

Equivalent method on ndarray object.

### Notes

`a.view()` is used two different ways:

`a.view(some_dtype)` or `a.view(dtype=some_dtype)` constructs a view of the array's memory with a different data-type. This can cause a reinterpretation of the bytes of memory.

`a.view(ndarray_subclass)` or `a.view(type=ndarray_subclass)` just returns an instance of *ndarray_subclass* that looks at the same array (same shape, dtype, etc.) This does not cause a reinterpretation of the memory.

If *`fill_value`* is not specified, but *`dtype`* is specified (and is not an ndarray sub-class), the *`fill_value`* of the MaskedArray will be reset. If neither *`fill_value`* nor *`dtype`* are specified (or if *`dtype`* is an ndarray sub-class), then the fill value is preserved. Finally, if *`fill_value`* is specified, but *`dtype`* is not, the fill value is set to the specified value.

For `a.view(some_dtype)`, if `some_dtype` has a different number of bytes per entry than the previous dtype (for example, converting a regular array to a structured array), then the behavior of the view cannot be predicted just from the superficial appearance of `a` (shown by `print(a)`). It also depends on exactly how `a` is stored in memory. Therefore if `a` is C-ordered versus fortran-ordered, versus defined as a slice or transpose, etc., the view may give different results.

method

`MaskedArray.`**`astype`**(*dtype*, *order='K'*, *casting='unsafe'*, *subok=True*, *copy=True*)

Copy of the array, cast to a specified type.

**Parameters**

**dtype**

[str or dtype] Typecode or data-type to which the array is cast.

**order**

[{'C', 'F', 'A', 'K'}, optional] Controls the memory layout order of the result. 'C' means C order, 'F' means Fortran order, 'A' means 'F' order if all the arrays are Fortran contiguous, 'C' order otherwise, and 'K' means as close to the order the array elements appear in memory as possible. Default is 'K'.

**casting**

[{'no', 'equiv', 'safe', 'same_kind', 'unsafe'}, optional] Controls what kind of data casting may occur. Defaults to 'unsafe' for backwards compatibility.

- 'no' means the data types should not be cast at all.
- 'equiv' means only byte-order changes are allowed.
- 'safe' means only casts which can preserve values are allowed.
- 'same_kind' means only safe casts or casts within a kind, like float64 to float32, are allowed.
- 'unsafe' means any data conversions may be done.

**subok**

[bool, optional] If True, then sub-classes will be passed-through (default), otherwise the returned array will be forced to be a base-class array.

**copy**

[bool, optional] By default, astype always returns a newly allocated array. If this is set to false, and the `dtype`, *order*, and *subok* requirements are satisfied, the input array is returned instead of a copy.

**Returns**

**arr_t**

[ndarray] Unless `copy` is False and the other conditions for returning the input array are satisfied (see description for `copy` input parameter), *arr_t* is a new array of the same shape as the input array, with dtype, order given by `dtype`, *order*.

**Raises**

**ComplexWarning**

When casting from complex to float or int. To avoid this, one should use `a.real.astype(t)`.

### Notes

Changed in version 1.17.0: Casting between a simple data type and a structured one is possible only for "unsafe" casting. Casting to multiple fields is allowed, but casting from multiple fields is not.

Changed in version 1.9.0: Casting from numeric to string types in 'safe' casting mode requires that the string dtype length is long enough to store the max integer/float value converted.

### Examples

```
>>> x = np.array([1, 2, 2.5])
>>> x
array([1. ,  2. ,  2.5])
```

```
>>> x.astype(int)
array([1, 2, 2])
```

method

`MaskedArray.`**`byteswap`**`(`*inplace=False*`)`

Swap the bytes of the array elements

Toggle between low-endian and big-endian data representation by returning a byteswapped array, optionally swapped in-place. Arrays of byte-strings are not swapped. The real and imaginary parts of a complex number are swapped individually.

**Parameters**

**inplace**

[bool, optional] If `True`, swap bytes in-place, default is `False`.

**Returns**

**out**

[ndarray] The byteswapped array. If *inplace* is `True`, this is a view to self.

### Examples

```
>>> A = np.array([1, 256, 8755], dtype=np.int16)
>>> list(map(hex, A))
['0x1', '0x100', '0x2233']
>>> A.byteswap(inplace=True)
array([  256,     1, 13090], dtype=int16)
>>> list(map(hex, A))
['0x100', '0x1', '0x3322']
```

Arrays of byte-strings are not swapped

```
>>> A = np.array([b'ceg', b'fac'])
>>> A.byteswap()
array([b'ceg', b'fac'], dtype='|S3')
```

**`A.newbyteorder().byteswap()` produces an array with the same values**

but different representation in memory

```
>>> A = np.array([1, 2, 3])
>>> A.view(np.uint8)
array([1, 0, 0, 0, 0, 0, 0, 0, 2, 0, 0, 0, 0, 0, 0, 0, 3, 0, 0, 0, 0, 0,
       0, 0], dtype=uint8)
>>> A.newbyteorder().byteswap(inplace=True)
array([1, 2, 3])
>>> A.view(np.uint8)
array([0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 2, 0, 0, 0, 0, 0, 0,
       0, 3], dtype=uint8)
```

method

`MaskedArray.`**`compressed`**`(`*self*`)`

Return all the non-masked data as a 1-D array.

**Returns**

**data**

[ndarray] A new *ndarray* holding the non-masked data is returned.

### Notes

The result is **not** a MaskedArray!

### Examples

```
>>> x = np.ma.array(np.arange(5), mask=[0]*2 + [1]*3)
>>> x.compressed()
array([0, 1])
>>> type(x.compressed())
<class 'numpy.ndarray'>
```

method

`MaskedArray.`**`filled`**(*self*, *fill_value=None*)

Return a copy of self, with masked values filled with a given value. **However**, if there are no masked values to fill, self will be returned instead as an ndarray.

**Parameters**

**fill_value**

[array_like, optional] The value to use for invalid entries. Can be scalar or non-scalar. If non-scalar, the resulting ndarray must be broadcastable over input array. Default is None, in which case, the *`fill_value`* attribute of the array is used instead.

**Returns**

**filled_array**

[ndarray] A copy of `self` with invalid entries replaced by *fill_value* (be it the function argument or the attribute of `self`), or `self` itself as an ndarray if there are no invalid entries to be replaced.

### Notes

The result is **not** a MaskedArray!

### Examples

```
>>> x = np.ma.array([1,2,3,4,5], mask=[0,0,1,0,1], fill_value=-999)
>>> x.filled()
array([   1,    2, -999,    4, -999])
>>> x.filled(fill_value=1000)
array([   1,    2, 1000,    4, 1000])
>>> type(x.filled())
<class 'numpy.ndarray'>
```

Subclassing is preserved. This means that if, e.g., the data part of the masked array is a recarray, *`filled`* returns a recarray:

```
>>> x = np.array([(-1, 2), (-3, 4)], dtype='i8,i8').view(np.recarray)
>>> m = np.ma.array(x, mask=[(True, False), (False, True)])
>>> m.filled()
rec.array([(999999,      2), (    -3, 999999)],
          dtype=[('f0', '<i8'), ('f1', '<i8')])
```

method

`MaskedArray.`**`tofile`**(*self*, *fid*, *sep=''*, *format='%s'*)

Save a masked array to a file in binary format.

**Warning:** This function is not implemented yet.

**Raises**

**NotImplementedError**

When *`tofile`* is called.

method

`MaskedArray.`**`toflex`**`(`*self*`)`

Transforms a masked array into a flexible-type array.

The flexible type array that is returned will have two fields:

- the `_data` field stores the `_data` part of the array.
- the `_mask` field stores the `_mask` part of the array.

**Parameters**

**None**

**Returns**

**record**

[ndarray] A new flexible-type *ndarray* with two fields: the first element containing a value, the second element containing the corresponding mask boolean. The returned record shape matches self.shape.

## Notes

A side-effect of transforming a masked array into a flexible *ndarray* is that meta information (`fill_value`, …) will be lost.

## Examples

```
>>> x = np.ma.array([[1,2,3],[4,5,6],[7,8,9]], mask=[0] + [1,0]*4)
>>> x
masked_array(
  data=[[1, --, 3],
        [--, 5, --],
        [7, --, 9]],
  mask=[[False,  True, False],
        [ True, False,  True],
        [False,  True, False]],
  fill_value=999999)
>>> x.toflex()
array([[(1, False), (2,  True), (3, False)],
       [(4,  True), (5, False), (6,  True)],
       [(7, False), (8,  True), (9, False)]],
      dtype=[('_data', '<i8'), ('_mask', '?')])
```

method

`MaskedArray.`**`tolist`**`(`*self*, *fill_value=None*`)`

Return the data portion of the masked array as a hierarchical Python list.

Data items are converted to the nearest compatible Python type. Masked values are converted to *`fill_value`*. If *`fill_value`* is None, the corresponding entries in the output list will be `None`.

**Parameters**

**fill_value**

[scalar, optional] The value to use for invalid entries. Default is None.

**Returns**

**result**

[list] The Python list representation of the masked array.

## Examples

```
>>> x = np.ma.array([[1,2,3], [4,5,6], [7,8,9]], mask=[0] + [1,0]*4)
>>> x.tolist()
[[1, None, 3], [None, 5, None], [7, None, 9]]
>>> x.tolist(-999)
[[1, -999, 3], [-999, 5, -999], [7, -999, 9]]
```

method

`MaskedArray.`**`torecords`**`(`*self*`)`

Transforms a masked array into a flexible-type array.

The flexible type array that is returned will have two fields:

- the `_data` field stores the `_data` part of the array.
- the `_mask` field stores the `_mask` part of the array.

**Parameters**

**None**

**Returns**

**record**

[ndarray] A new flexible-type *ndarray* with two fields: the first element containing a value, the second element containing the corresponding mask boolean. The returned record shape matches self.shape.

## Notes

A side-effect of transforming a masked array into a flexible *ndarray* is that meta information (`fill_value`, …) will be lost.

## Examples

```
>>> x = np.ma.array([[1,2,3],[4,5,6],[7,8,9]], mask=[0] + [1,0]*4)
>>> x
masked_array(
  data=[[1, --, 3],
        [--, 5, --],
        [7, --, 9]],
  mask=[[False,  True, False],
        [ True, False,  True],
        [False,  True, False]],
  fill_value=999999)
>>> x.toflex()
array([[(1, False), (2,  True), (3, False)],
       [(4,  True), (5, False), (6,  True)],
       [(7, False), (8,  True), (9, False)]],
      dtype=[('_data', '<i8'), ('_mask', '?')])
```

method

`MaskedArray.`**`tostring`**(*self*, *fill_value=None*, *order='C'*)

A compatibility alias for `tobytes`, with exactly the same behavior.

Despite its name, it returns *bytes* not *str*s.

Deprecated since version 1.19.0.

method

`MaskedArray.`**`tobytes`**(*self*, *fill_value=None*, *order='C'*)

Return the array data as a string containing the raw bytes in the array.

The array is filled with a fill value before the string conversion.

New in version 1.9.0.

**Parameters**

**fill_value**

[scalar, optional] Value used to fill in the masked values. Default is None, in which case `MaskedArray.fill_value` is used.

**order**

[{'C','F','A'}, optional] Order of the data item in the copy. Default is 'C'.

- 'C' – C order (row major).
- 'F' – Fortran order (column major).
- 'A' – Any, current order of array.
- None – Same as 'A'.

**See also:**

`numpy.ndarray.tobytes`, `tolist`, `tofile`

### Notes

As for *ndarray.tobytes*, information about the shape, dtype, etc., but also about `fill_value`, will be lost.

### Examples

```
>>> x = np.ma.array(np.array([[1, 2], [3, 4]]), mask=[[0, 1], [1, 0]])
>>> x.tobytes()
b'\x01\x00\x00\x00\x00\x00\x00\x00?B\x0f\x00\x00\x00\x00\x00?B\x0f\x00\x00\x00\
↪x00\x00\x04\x00\x00\x00\x00\x00\x00\x00'
```

## Shape manipulation

For reshape, resize, and transpose, the single tuple argument may be replaced with `n` integers which will be interpreted as an n-tuple.

| | |
|---|---|
| `MaskedArray.flatten`([order]) | Return a copy of the array collapsed into one dimension. |
| `MaskedArray.ravel`(self[, order]) | Returns a 1D version of self, as a view. |
| `MaskedArray.reshape`(self, *s, **kwargs) | Give a new shape to the array without changing its data. |
| `MaskedArray.resize`(self, newshape[, …]) | |
| `MaskedArray.squeeze`([axis]) | Remove single-dimensional entries from the shape of *a*. |
| `MaskedArray.swapaxes`(axis1, axis2) | Return a view of the array with *axis1* and *axis2* interchanged. |
| `MaskedArray.transpose`(*axes) | Returns a view of the array with axes transposed. |
| `MaskedArray.T` | |

method

`MaskedArray.`**`flatten`**(*order='C'*)

Return a copy of the array collapsed into one dimension.

**Parameters**

**order**

[{'C', 'F', 'A', 'K'}, optional] 'C' means to flatten in row-major (C-style) order. 'F' means to flatten in column-major (Fortran- style) order. 'A' means to flatten in column-major order if *a* is Fortran *contiguous* in memory, row-major order otherwise. 'K' means to flatten *a* in the order the elements occur in memory. The default is 'C'.

**Returns**

**y**

[ndarray] A copy of the input array, flattened to one dimension.

**See also:**

**`ravel`**

Return a flattened array.

**`flat`**

A 1-D flat iterator over the array.

### Examples

```
>>> a = np.array([[1,2], [3,4]])
>>> a.flatten()
array([1, 2, 3, 4])
>>> a.flatten('F')
array([1, 3, 2, 4])
```

method

`MaskedArray.`**`ravel`**(*self*, *order='C'*)

Returns a 1D version of self, as a view.

**Parameters**

**order**

[{'C', 'F', 'A', 'K'}, optional] The elements of *a* are read using this index order. 'C' means to index the elements in C-like order, with the last axis index changing fastest, back to the first axis index changing slowest. 'F' means to index the elements in Fortran-like index order, with the first index changing fastest, and the last index changing slowest. Note that the 'C' and 'F' options take no account of the memory layout of the underlying array, and only refer to the order of axis indexing. 'A' means to read the elements in Fortran-like index order if *m* is Fortran *contiguous* in memory, C-like order otherwise. 'K' means to read the elements in the order they occur in memory, except for reversing the data when strides are negative. By default, 'C' index order is used.

**Returns**

**MaskedArray**

Output view is of shape `(self.size,)` (or `(np.ma.product(self.shape),)`).

### Examples

```
>>> x = np.ma.array([[1,2,3],[4,5,6],[7,8,9]], mask=[0] + [1,0]*4)
>>> x
masked_array(
  data=[[1, --, 3],
        [--, 5, --],
        [7, --, 9]],
  mask=[[False,  True, False],
        [ True, False,  True],
        [False,  True, False]],
  fill_value=999999)
>>> x.ravel()
masked_array(data=[1, --, 3, --, 5, --, 7, --, 9],
             mask=[False,  True, False,  True, False,  True, False,  True,
                   False],
       fill_value=999999)
```

method

`MaskedArray.`**`reshape`**(*self*, **s*, ***kwargs*)

Give a new shape to the array without changing its data.

Returns a masked array containing the same data, but with a new shape. The result is a view on the original array; if this is not possible, a ValueError is raised.

**Parameters**

**shape**

[int or tuple of ints] The new shape should be compatible with the original shape. If an integer is supplied, then the result will be a 1-D array of that length.

**order**

[{'C', 'F'}, optional] Determines whether the array data should be viewed as in C (row-major) or FORTRAN (column-major) order.

**Returns**

**reshaped_array**

[array] A new view on the array.

**See also:**

***`reshape`***

Equivalent function in the masked array module.

***`numpy.ndarray.reshape`***

Equivalent method on ndarray object.

***`numpy.reshape`***

Equivalent function in the NumPy module.

## Notes

The reshaping operation cannot guarantee that a copy will not be made, to modify the shape in place, use `a.shape = s`

## Examples

```
>>> x = np.ma.array([[1,2],[3,4]], mask=[1,0,0,1])
>>> x
masked_array(
  data=[[--, 2],
        [3, --]],
  mask=[[ True, False],
        [False,  True]],
  fill_value=999999)
>>> x = x.reshape((4,1))
>>> x
masked_array(
  data=[[--],
        [2],
```

(continues on next page)

(continued from previous page)

```
        [3],
        [--]],
  mask=[[ True],
        [False],
        [False],
        [ True]],
  fill_value=999999)
```

method

`MaskedArray.`**`resize`**`(`*self*, *newshape*, *refcheck=True*, *order=False*`)`

> **Warning:** This method does nothing, except raise a ValueError exception. A masked array does not own its data and therefore cannot safely be resized in place. Use the *`numpy.ma.resize`* function instead.

This method is difficult to implement safely and may be deprecated in future releases of NumPy.

method

`MaskedArray.`**`squeeze`**`(`*axis=None*`)`

Remove single-dimensional entries from the shape of *a*.

Refer to *`numpy.squeeze`* for full documentation.

**See also:**

***`numpy.squeeze`***

equivalent function

method

`MaskedArray.`**`swapaxes`**`(`*axis1*, *axis2*`)`

Return a view of the array with *axis1* and *axis2* interchanged.

Refer to *`numpy.swapaxes`* for full documentation.

**See also:**

***`numpy.swapaxes`***

equivalent function

method

`MaskedArray.`**`transpose`**`(`**axes*`)`

Returns a view of the array with axes transposed.

For a 1-D array this has no effect, as a transposed vector is simply the same vector. To convert a 1-D array into a 2D column vector, an additional dimension must be added. *np.atleast2d(a).T* achieves this, as does *a[:, np.newaxis]*. For a 2-D array, this is a standard matrix transpose. For an n-D array, if axes are given, their order indicates how the axes are permuted (see Examples). If axes are not provided and `a.shape = (i[0], i[1], ... i[n-2], i[n-1])`, then `a.transpose().shape = (i[n-1], i[n-2], ... i[1], i[0])`.

**Parameters**

**axes**

[None, tuple of ints, or *n* ints]

- None or no argument: reverses the order of the axes.
- tuple of ints: *i* in the *j*-th place in the tuple means *a*'s *i*-th axis becomes *a.transpose()*'s *j*-th axis.
- *n* ints: same as an n-tuple of the same ints (this form is intended simply as a "convenience" alternative to the tuple form)

**Returns**

**out**

[ndarray] View of *a*, with axes suitably permuted.

**See also:**

**`ndarray.T`**

Array property returning the array transposed.

**`ndarray.reshape`**

Give a new shape to an array without changing its data.

### Examples

```
>>> a = np.array([[1, 2], [3, 4]])
>>> a
array([[1, 2],
       [3, 4]])
>>> a.transpose()
array([[1, 3],
       [2, 4]])
>>> a.transpose((1, 0))
array([[1, 3],
       [2, 4]])
>>> a.transpose(1, 0)
array([[1, 3],
       [2, 4]])
```

property

**property** `MaskedArray.`**`T`**

### Item selection and manipulation

For array methods that take an `axis` keyword, it defaults to None. If axis is None, then the array is treated as a 1-D array. Any other value for `axis` represents the dimension along which the operation should proceed.

| | |
|---|---|
| *`MaskedArray.argmax`*(self[, axis, fill_value, out]) | Returns array of indices of the maximum values along the given axis. |
| *`MaskedArray.argmin`*(self[, axis, fill_value, out]) | Return array of indices to the minimum values along the given axis. |
| *`MaskedArray.argsort`*(self[, axis, kind, …]) | Return an ndarray of indices that sort the array along the specified axis. |
| *`MaskedArray.choose`*(choices[, out, mode]) | Use an index array to construct a new array from a set of choices. |
| *`MaskedArray.compress`*(self, condition[, …]) | Return *a* where condition is `True`. |
| *`MaskedArray.diagonal`*([offset, axis1, axis2]) | Return specified diagonals. |
| *`MaskedArray.fill`*(value) | Fill the array with a scalar value. |
| *`MaskedArray.item`*(*args) | Copy an element of an array to a standard Python scalar and return it. |
| *`MaskedArray.nonzero`*(self) | Return the indices of unmasked elements that are not zero. |
| *`MaskedArray.put`*(self, indices, values[, mode]) | Set storage-indexed locations to corresponding values. |
| *`MaskedArray.repeat`*(repeats[, axis]) | Repeat elements of an array. |
| *`MaskedArray.searchsorted`*(v[, side, sorter]) | Find indices where elements of v should be inserted in a to maintain order. |
| *`MaskedArray.sort`*(self[, axis, kind, order, …]) | Sort the array, in-place |
| *`MaskedArray.take`*(self, indices[, axis, out, …]) | |

method

`MaskedArray.`**`argmax`**(*self*, *axis=None*, *fill_value=None*, *out=None*)

Returns array of indices of the maximum values along the given axis. Masked values are treated as if they had the value fill_value.

**Parameters**

**axis**

[{None, integer}] If None, the index is into the flattened array, otherwise along the specified axis

**fill_value**

[{var}, optional] Value used to fill in the masked values. If None, the output of maximum_fill_value(self._data) is used instead.

**out**

[{None, array}, optional] Array into which the result can be placed. Its type is preserved and it must be of the right shape to hold the output.

**Returns**

**index_array**

[{integer_array}]

### Examples

```
>>> a = np.arange(6).reshape(2,3)
>>> a.argmax()
5
>>> a.argmax(0)
array([1, 1, 1])
>>> a.argmax(1)
array([2, 2])
```

method

`MaskedArray.`**`argmin`**(*self*, *axis=None*, *fill_value=None*, *out=None*)

Return array of indices to the minimum values along the given axis.

**Parameters**

**axis**

[{None, integer}] If None, the index is into the flattened array, otherwise along the specified axis

**fill_value**

[{var}, optional] Value used to fill in the masked values. If None, the output of minimum_fill_value(self._data) is used instead.

**out**

[{None, array}, optional] Array into which the result can be placed. Its type is preserved and it must be of the right shape to hold the output.

**Returns**

**ndarray or scalar**

If multi-dimension input, returns a new ndarray of indices to the minimum values along the given axis. Otherwise, returns a scalar of index to the minimum values along the given axis.

### Examples

```
>>> x = np.ma.array(np.arange(4), mask=[1,1,0,0])
>>> x.shape = (2,2)
>>> x
masked_array(
  data=[[--, --],
        [2, 3]],
  mask=[[ True,  True],
        [False, False]],
  fill_value=999999)
>>> x.argmin(axis=0, fill_value=-1)
array([0, 0])
>>> x.argmin(axis=0, fill_value=9)
array([1, 1])
```

method

`MaskedArray.`**`argsort`**(*self*, *axis=<no value>*, *kind=None*, *order=None*, *endwith=True*, *fill_value=None*)

Return an ndarray of indices that sort the array along the specified axis. Masked values are filled beforehand to *`fill_value`*.

**Parameters**

**axis**

[int, optional] Axis along which to sort. If None, the default, the flattened array is used.

Changed in version 1.13.0: Previously, the default was documented to be -1, but that was in error. At some future date, the default will change to -1, as originally intended. Until then, the axis should be given explicitly when `arr.ndim > 1`, to avoid a FutureWarning.

**kind**

[{'quicksort', 'mergesort', 'heapsort', 'stable'}, optional] The sorting algorithm used.

**order**

[list, optional] When *a* is an array with fields defined, this argument specifies which fields to compare first, second, etc. Not all fields need be specified.

**endwith**

[{True, False}, optional] Whether missing values (if any) should be treated as the largest values (True) or the smallest values (False) When the array contains unmasked values at the same extremes of the datatype, the ordering of these values and the masked values is undefined.

**fill_value**

[{var}, optional] Value used internally for the masked values. If `fill_value` is not None, it supersedes `endwith`.

**Returns**

**index_array**

[ndarray, int] Array of indices that sort *a* along the specified axis. In other words, `a[index_array]` yields a sorted *a*.

**See also:**

***`MaskedArray.sort`***

Describes sorting algorithms used.

**`lexsort`**

Indirect stable sort with multiple keys.

***`numpy.ndarray.sort`***

Inplace sort.

### Notes

See `sort` for notes on the different sorting algorithms.

### Examples

```
>>> a = np.ma.array([3,2,1], mask=[False, False, True])
>>> a
masked_array(data=[3, 2, --],
             mask=[False, False,  True],
       fill_value=999999)
>>> a.argsort()
array([1, 0, 2])
```

method

`MaskedArray.`**`choose`**(*choices*, *out=None*, *mode='raise'*)

Use an index array to construct a new array from a set of choices.

Refer to `numpy.choose` for full documentation.

**See also:**

**`numpy.choose`**

equivalent function

method

`MaskedArray.`**`compress`**(*self*, *condition*, *axis=None*, *out=None*)

Return *a* where condition is `True`.

If condition is a `MaskedArray`, missing values are considered as `False`.

**Parameters**

**condition**

[var] Boolean 1-d array selecting which entries to return. If len(condition) is less than the size of a along the axis, then output is truncated to length of condition array.

**axis**

[{None, int}, optional] Axis along which the operation must be performed.

**out**

[{None, ndarray}, optional] Alternative output array in which to place the result. It must have the same shape as the expected output but the type will be cast if necessary.

**Returns**

**result**

[MaskedArray] A `MaskedArray` object.

### Notes

Please note the difference with `compressed` ! The output of `compress` has a mask, the output of `compressed` does not.

### Examples

```
>>> x = np.ma.array([[1,2,3],[4,5,6],[7,8,9]], mask=[0] + [1,0]*4)
>>> x
masked_array(
  data=[[1, --, 3],
        [--, 5, --],
        [7, --, 9]],
  mask=[[False,  True, False],
        [ True, False,  True],
        [False,  True, False]],
  fill_value=999999)
>>> x.compress([1, 0, 1])
masked_array(data=[1, 3],
             mask=[False, False],
       fill_value=999999)
```

```
>>> x.compress([1, 0, 1], axis=1)
masked_array(
  data=[[1, 3],
        [--, --],
        [7, 9]],
  mask=[[False, False],
        [ True,  True],
        [False, False]],
  fill_value=999999)
```

method

`MaskedArray.`**`diagonal`**`(`*offset=0*, *axis1=0*, *axis2=1*`)`

Return specified diagonals. In NumPy 1.9 the returned array is a read-only view instead of a copy as in previous NumPy versions. In a future version the read-only restriction will be removed.

Refer to `numpy.diagonal` for full documentation.

**See also:**

**`numpy.diagonal`**

equivalent function

method

`MaskedArray.`**`fill`**`(`*value*`)`

Fill the array with a scalar value.

**Parameters**

**value**

[scalar] All elements of *a* will be assigned this value.

### Examples

```
>>> a = np.array([1, 2])
>>> a.fill(0)
>>> a
array([0, 0])
>>> a = np.empty(2)
>>> a.fill(1)
>>> a
array([1.,  1.])
```

method

`MaskedArray.`**`item`**`(`**args*`)`

Copy an element of an array to a standard Python scalar and return it.

**Parameters**

***args**

[Arguments (variable number and type)]

- none: in this case, the method only works for arrays with one element (*a.size == 1*), which element is copied into a standard Python scalar object and returned.
- int_type: this argument is interpreted as a flat index into the array, specifying which element to copy and return.
- tuple of int_types: functions as does a single int_type argument, except that the argument is interpreted as an nd-index into the array.

**Returns**

**z**

[Standard Python scalar object] A copy of the specified element of the array as a suitable Python scalar

### Notes

When the data type of *a* is longdouble or clongdouble, item() returns a scalar array object because there is no available Python scalar that would not lose information. Void arrays return a buffer object for item(), unless fields are defined, in which case a tuple is returned.

`item` is very similar to a[args], except, instead of an array scalar, a standard Python scalar is returned. This can be useful for speeding up access to elements of the array and doing arithmetic on elements of the array using Python's optimized math.

### Examples

```
>>> np.random.seed(123)
>>> x = np.random.randint(9, size=(3, 3))
>>> x
array([[2, 2, 6],
       [1, 3, 6],
       [1, 0, 1]])
>>> x.item(3)
1
>>> x.item(7)
0
>>> x.item((0, 1))
2
>>> x.item((2, 2))
1
```

method

`MaskedArray.`**`nonzero`**`(`*self*`)`

Return the indices of unmasked elements that are not zero.

Returns a tuple of arrays, one for each dimension, containing the indices of the non-zero elements in that dimension. The corresponding non-zero values can be obtained with:

```
a[a.nonzero()]
```

To group the indices by element, rather than dimension, use instead:

```
np.transpose(a.nonzero())
```

The result of this is always a 2d array, with a row for each non-zero element.

**Parameters**

**None**

**Returns**

**tuple_of_arrays**

[tuple] Indices of elements that are non-zero.

**See also:**

***`numpy.nonzero`***

Function operating on ndarrays.

**`flatnonzero`**

Return indices that are non-zero in the flattened version of the input array.

***`numpy.ndarray.nonzero`***

Equivalent ndarray method.

**`count_nonzero`**

Counts the number of non-zero elements in the input array.

### Examples

```
>>> import numpy.ma as ma
>>> x = ma.array(np.eye(3))
>>> x
masked_array(
  data=[[1., 0., 0.],
        [0., 1., 0.],
        [0., 0., 1.]],
  mask=False,
  fill_value=1e+20)
>>> x.nonzero()
(array([0, 1, 2]), array([0, 1, 2]))
```

Masked elements are ignored.

```
>>> x[1, 1] = ma.masked
>>> x
masked_array(
  data=[[1.0, 0.0, 0.0],
        [0.0, --, 0.0],
        [0.0, 0.0, 1.0]],
  mask=[[False, False, False],
        [False,  True, False],
        [False, False, False]],
  fill_value=1e+20)
>>> x.nonzero()
(array([0, 2]), array([0, 2]))
```

Indices can also be grouped by element.

```
>>> np.transpose(x.nonzero())
array([[0, 0],
       [2, 2]])
```

A common use for `nonzero` is to find the indices of an array, where a condition is True. Given an array *a*, the condition $a > 3$ is a boolean array and since False is interpreted as 0, ma.nonzero(a > 3) yields the indices of the *a* where the condition is true.

```
>>> a = ma.array([[1,2,3],[4,5,6],[7,8,9]])
>>> a > 3
masked_array(
  data=[[False, False, False],
        [ True,  True,  True],
        [ True,  True,  True]],
  mask=False,
  fill_value=True)
>>> ma.nonzero(a > 3)
(array([1, 1, 1, 2, 2, 2]), array([0, 1, 2, 0, 1, 2]))
```

The `nonzero` method of the condition array can also be called.

```
>>> (a > 3).nonzero()
(array([1, 1, 1, 2, 2, 2]), array([0, 1, 2, 0, 1, 2]))
```

method

`MaskedArray`**.put**(*self*, *indices*, *values*, *mode='raise'*)

Set storage-indexed locations to corresponding values.

Sets self._data.flat[n] = values[n] for each n in indices. If *values* is shorter than `indices` then it will repeat. If *values* has some masked values, the initial mask is updated in consequence, else the corresponding values are unmasked.

**Parameters**

**indices**

[1-D array_like] Target indices, interpreted as integers.

**values**

[array_like] Values to place in self._data copy at target indices.

**mode**

[{'raise', 'wrap', 'clip'}, optional] Specifies how out-of-bounds indices will behave. 'raise' : raise an error. 'wrap' : wrap around. 'clip' : clip to the range.

## Notes

*values* can be a scalar or length 1 array.

## Examples

```
>>> x = np.ma.array([[1,2,3],[4,5,6],[7,8,9]], mask=[0] + [1,0]*4)
>>> x
masked_array(
  data=[[1, --, 3],
        [--, 5, --],
        [7, --, 9]],
  mask=[[False,  True, False],
        [ True, False,  True],
        [False,  True, False]],
  fill_value=999999)
>>> x.put([0,4,8],[10,20,30])
>>> x
masked_array(
  data=[[10, --, 3],
        [--, 20, --],
        [7, --, 30]],
  mask=[[False,  True, False],
        [ True, False,  True],
        [False,  True, False]],
  fill_value=999999)
```

```
>>> x.put(4,999)
>>> x
masked_array(
  data=[[10, --, 3],
        [--, 999, --],
        [7, --, 30]],
  mask=[[False,  True, False],
        [ True, False,  True],
```

(continues on next page)

(continued from previous page)

```
          [False,  True, False]],
  fill_value=999999)
```

method

`MaskedArray.`**`repeat`**(*repeats*, *axis=None*)

Repeat elements of an array.

Refer to `numpy.repeat` for full documentation.

**See also:**

***`numpy.repeat`***

equivalent function

method

`MaskedArray.`**`searchsorted`**(*v*, *side='left'*, *sorter=None*)

Find indices where elements of v should be inserted in a to maintain order.

For full documentation, see `numpy.searchsorted`

**See also:**

***`numpy.searchsorted`***

equivalent function

method

`MaskedArray.`**`sort`**(*self*, *axis=-1*, *kind=None*, *order=None*, *endwith=True*, *fill_value=None*)

Sort the array, in-place

**Parameters**

**a**

[array_like] Array to be sorted.

**axis**

[int, optional] Axis along which to sort. If None, the array is flattened before sorting. The default is -1, which sorts along the last axis.

**kind**

[{'quicksort', 'mergesort', 'heapsort', 'stable'}, optional] The sorting algorithm used.

**order**

[list, optional] When *a* is a structured array, this argument specifies which fields to compare first, second, and so on. This list does not need to include all of the fields.

**endwith**

[{True, False}, optional] Whether missing values (if any) should be treated as the largest values (True) or the smallest values (False) When the array contains unmasked values sorting at the same extremes of the datatype, the ordering of these values and the masked values is undefined.

**fill_value**

[{var}, optional] Value used internally for the masked values. If `fill_value` is not None, it supersedes `endwith`.

**Returns**

**sorted_array**

[ndarray] Array of the same type and shape as *a*.

**See also:**

***`numpy.ndarray.sort`***

Method to sort an array in-place.

***`argsort`***

Indirect sort.

**`lexsort`**

Indirect stable sort on multiple keys.

***`searchsorted`***

Find elements in a sorted array.

## Notes

See `sort` for notes on the different sorting algorithms.

## Examples

```
>>> a = np.ma.array([1, 2, 5, 4, 3],mask=[0, 1, 0, 1, 0])
>>> # Default
>>> a.sort()
>>> a
masked_array(data=[1, 3, 5, --, --],
             mask=[False, False, False,  True,  True],
       fill_value=999999)
```

```
>>> a = np.ma.array([1, 2, 5, 4, 3],mask=[0, 1, 0, 1, 0])
>>> # Put missing values in the front
>>> a.sort(endwith=False)
>>> a
masked_array(data=[--, --, 1, 3, 5],
             mask=[ True,  True, False, False, False],
       fill_value=999999)
```

```
>>> a = np.ma.array([1, 2, 5, 4, 3],mask=[0, 1, 0, 1, 0])
>>> # fill_value takes over endwith
>>> a.sort(endwith=False, fill_value=3)
>>> a
masked_array(data=[1, --, --, 3, 5],
```

(continues on next page)

(continued from previous page)

```
             mask=[False,  True,  True, False, False],
       fill_value=999999)
```

method

`MaskedArray.`**`take`**(*self*, *indices*, *axis=None*, *out=None*, *mode='raise'*)

## Pickling and copy

| | |
|---|---|
| *`MaskedArray.copy`*([order]) | Return a copy of the array. |
| *`MaskedArray.dump`*(file) | Dump a pickle of the array to the specified file. |
| *`MaskedArray.dumps`*() | Returns the pickle of the array as a string. |

method

`MaskedArray.`**`copy`**(*order='C'*)

Return a copy of the array.

**Parameters**

**order**

[{'C', 'F', 'A', 'K'}, optional] Controls the memory layout of the copy. 'C' means C-order, 'F' means F-order, 'A' means 'F' if *a* is Fortran contiguous, 'C' otherwise. 'K' means match the layout of *a* as closely as possible. (Note that this function and *`numpy.copy`* are very similar, but have different default values for their order= arguments.)

**See also:**

*`numpy.copy`*, *`numpy.copyto`*

### Examples

```
>>> x = np.array([[1,2,3],[4,5,6]], order='F')
```

```
>>> y = x.copy()
```

```
>>> x.fill(0)
```

```
>>> x
array([[0, 0, 0],
       [0, 0, 0]])
```

```
>>> y
array([[1, 2, 3],
       [4, 5, 6]])
```

```
>>> y.flags['C_CONTIGUOUS']
True
```

method

`MaskedArray.`**`dump`**`(`*file*`)`

Dump a pickle of the array to the specified file. The array can be read back with pickle.load or numpy.load.

**Parameters**

**file**

[str or Path] A string naming the dump file.

Changed in version 1.17.0: `pathlib.Path` objects are now accepted.

method

`MaskedArray.`**`dumps`**`()`

Returns the pickle of the array as a string. pickle.loads or numpy.loads will convert the string back to an array.

**Parameters**

**None**

## Calculations

| | |
|---|---|
| *`MaskedArray.all`*(self[, axis, out, keepdims]) | Returns True if all elements evaluate to True. |
| *`MaskedArray.anom`*(self[, axis, dtype]) | Compute the anomalies (deviations from the arithmetic mean) along the given axis. |
| *`MaskedArray.any`*(self[, axis, out, keepdims]) | Returns True if any of the elements of *a* evaluate to True. |
| *`MaskedArray.clip`*([min, max, out]) | Return an array whose values are limited to `[min, max]`. |
| *`MaskedArray.conj`*() | Complex-conjugate all elements. |
| *`MaskedArray.conjugate`*() | Return the complex conjugate, element-wise. |
| *`MaskedArray.cumprod`*(self[, axis, dtype, out]) | Return the cumulative product of the array elements over the given axis. |
| *`MaskedArray.cumsum`*(self[, axis, dtype, out]) | Return the cumulative sum of the array elements over the given axis. |
| *`MaskedArray.max`*(self[, axis, out, …]) | Return the maximum along a given axis. |
| *`MaskedArray.mean`*(self[, axis, dtype, out, …]) | Returns the average of the array elements along given axis. |
| *`MaskedArray.min`*(self[, axis, out, …]) | Return the minimum along a given axis. |
| *`MaskedArray.prod`*(self[, axis, dtype, out, …]) | Return the product of the array elements over the given axis. |
| *`MaskedArray.product`*(self[, axis, dtype, …]) | Return the product of the array elements over the given axis. |
| *`MaskedArray.ptp`*(self[, axis, out, …]) | Return (maximum - minimum) along the given dimension (i.e. |
| *`MaskedArray.round`*(self[, decimals, out]) | Return each element rounded to the given number of decimals. |
| *`MaskedArray.std`*(self[, axis, dtype, out, …]) | Returns the standard deviation of the array elements along given axis. |
| *`MaskedArray.sum`*(self[, axis, dtype, out, …]) | Return the sum of the array elements over the given axis. |
| *`MaskedArray.trace`*([offset, axis1, axis2, …]) | Return the sum along diagonals of the array. |
| *`MaskedArray.var`*(self[, axis, dtype, out, …]) | Compute the variance along the specified axis. |

method

`MaskedArray`.**`all`**(*self*, *axis=None*, *out=None*, *keepdims=<no value>*)

Returns True if all elements evaluate to True.

The output array is masked where all the values along the given axis are masked: if the output would have been a scalar and that all the values are masked, then the output is *`masked`*.

Refer to *`numpy.all`* for full documentation.

**See also:**

***`numpy.ndarray.all`***

corresponding function for ndarrays

***`numpy.all`***

equivalent function

### Examples

```
>>> np.ma.array([1,2,3]).all()
True
>>> a = np.ma.array([1,2,3], mask=True)
>>> (a.all() is np.ma.masked)
True
```

method

`MaskedArray`.**`anom`**(*self*, *axis=None*, *dtype=None*)

Compute the anomalies (deviations from the arithmetic mean) along the given axis.

Returns an array of anomalies, with the same shape as the input and where the arithmetic mean is computed along the given axis.

**Parameters**

**axis**

[int, optional] Axis over which the anomalies are taken. The default is to use the mean of the flattened array as reference.

**dtype**

[dtype, optional]

**Type to use in computing the variance. For arrays of integer type**

the default is float32; for arrays of float types it is the same as the array type.

**See also:**

***`mean`***

Compute the mean of the array.

### Examples

```
>>> a = np.ma.array([1,2,3])
>>> a.anom()
masked_array(data=[-1.,  0.,  1.],
             mask=False,
       fill_value=1e+20)
```

method

`MaskedArray`.**any**(*self*, *axis=None*, *out=None*, *keepdims=<no value>*)

Returns True if any of the elements of *a* evaluate to True.

Masked values are considered as False during computation.

Refer to `numpy.any` for full documentation.

**See also:**

**`numpy.ndarray.any`**

corresponding function for ndarrays

**`numpy.any`**

equivalent function

method

`MaskedArray`.**clip**(*min=None*, *max=None*, *out=None*, ***kwargs*)

Return an array whose values are limited to `[min, max]`. One of max or min must be given.

Refer to `numpy.clip` for full documentation.

**See also:**

**`numpy.clip`**

equivalent function

method

`MaskedArray`.**conj**()

Complex-conjugate all elements.

Refer to `numpy.conjugate` for full documentation.

**See also:**

**`numpy.conjugate`**

equivalent function

method

`MaskedArray`.**conjugate**()

Return the complex conjugate, element-wise.

Refer to `numpy.conjugate` for full documentation.

**See also:**

**`numpy.conjugate`**

equivalent function

method

`MaskedArray.`**`cumprod`**`(`*self*, *axis=None*, *dtype=None*, *out=None*`)`

Return the cumulative product of the array elements over the given axis.

Masked values are set to 1 internally during the computation. However, their position is saved, and the result will be masked at the same locations.

Refer to `numpy.cumprod` for full documentation.

**See also:**

**`numpy.ndarray.cumprod`**

corresponding function for ndarrays

**`numpy.cumprod`**

equivalent function

### Notes

The mask is lost if *out* is not a valid MaskedArray !

Arithmetic is modular when using integer types, and no error is raised on overflow.

method

`MaskedArray.`**`cumsum`**`(`*self*, *axis=None*, *dtype=None*, *out=None*`)`

Return the cumulative sum of the array elements over the given axis.

Masked values are set to 0 internally during the computation. However, their position is saved, and the result will be masked at the same locations.

Refer to `numpy.cumsum` for full documentation.

**See also:**

**`numpy.ndarray.cumsum`**

corresponding function for ndarrays

**`numpy.cumsum`**

equivalent function

### Notes

The mask is lost if *out* is not a valid `MaskedArray` !

Arithmetic is modular when using integer types, and no error is raised on overflow.

### Examples

```
>>> marr = np.ma.array(np.arange(10), mask=[0,0,0,1,1,1,0,0,0,0])
>>> marr.cumsum()
masked_array(data=[0, 1, 3, --, --, --, 9, 16, 24, 33],
             mask=[False, False, False,  True,  True,  True, False, False,
                   False, False],
       fill_value=999999)
```

method

`MaskedArray`**`.max`**(*self*, *axis=None*, *out=None*, *fill_value=None*, *keepdims=<no value>*)

Return the maximum along a given axis.

**Parameters**

**axis**

[{None, int}, optional] Axis along which to operate. By default, `axis` is None and the flattened input is used.

**out**

[array_like, optional] Alternative output array in which to place the result. Must be of the same shape and buffer length as the expected output.

**fill_value**

[{var}, optional] Value used to fill in the masked values. If None, use the output of maximum_fill_value().

**keepdims**

[bool, optional] If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the array.

**Returns**

**amax**

[array_like] New array holding the result. If `out` was specified, `out` is returned.

**See also:**

***`maximum_fill_value`***

Returns the maximum filling value for a given datatype.

method

`MaskedArray`**`.mean`**(*self*, *axis=None*, *dtype=None*, *out=None*, *keepdims=<no value>*)

Returns the average of the array elements along given axis.

Masked entries are ignored, and result elements which are not finite will be masked.

Refer to *`numpy.mean`* for full documentation.

**See also:**

***`numpy.ndarray.mean`***

corresponding function for ndarrays

***numpy.mean***

Equivalent function

***numpy.ma.average***

Weighted average.

### Examples

```
>>> a = np.ma.array([1,2,3], mask=[False, False, True])
>>> a
masked_array(data=[1, 2, --],
             mask=[False, False,  True],
       fill_value=999999)
>>> a.mean()
1.5
```

method

`MaskedArray`**`.min`**(*self*, *axis=None*, *out=None*, *fill_value=None*, *keepdims=<no value>*)

Return the minimum along a given axis.

**Parameters**

**axis**

[{None, int}, optional] Axis along which to operate. By default, `axis` is None and the flattened input is used.

**out**

[array_like, optional] Alternative output array in which to place the result. Must be of the same shape and buffer length as the expected output.

**fill_value**

[{var}, optional] Value used to fill in the masked values. If None, use the output of `minimum_fill_value`.

**keepdims**

[bool, optional] If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the array.

**Returns**

**amin**

[array_like] New array holding the result. If `out` was specified, `out` is returned.

**See also:**

**`minimum_fill_value`**

Returns the minimum filling value for a given datatype.

method

`MaskedArray`**.prod**(*self*, *axis=None*, *dtype=None*, *out=None*, *keepdims=<no value>*)

Return the product of the array elements over the given axis.

Masked elements are set to 1 internally for computation.

Refer to `numpy.prod` for full documentation.

**See also:**

**`numpy.ndarray.prod`**

corresponding function for ndarrays

**`numpy.prod`**

equivalent function

## Notes

Arithmetic is modular when using integer types, and no error is raised on overflow.

method

`MaskedArray`**.product**(*self*, *axis=None*, *dtype=None*, *out=None*, *keepdims=<no value>*)

Return the product of the array elements over the given axis.

Masked elements are set to 1 internally for computation.

Refer to `numpy.prod` for full documentation.

**See also:**

**`numpy.ndarray.prod`**

corresponding function for ndarrays

**`numpy.prod`**

equivalent function

## Notes

Arithmetic is modular when using integer types, and no error is raised on overflow.

method

`MaskedArray`**.ptp**(*self*, *axis=None*, *out=None*, *fill_value=None*, *keepdims=False*)

Return (maximum - minimum) along the given dimension (i.e. peak-to-peak value).

> **Warning:** `ptp` preserves the data type of the array. This means the return value for an input of signed integers with n bits (e.g. *np.int8*, *np.int16*, etc) is also a signed integer with n bits. In that case, peak-to-peak values greater than `2**(n-1)-1` will be returned as negative values. An example with a work-around is shown below.

**Parameters**

**axis**

[{None, int}, optional] Axis along which to find the peaks. If None (default) the flattened array is used.

**out**

[{None, array_like}, optional] Alternative output array in which to place the result. It must have the same shape and buffer length as the expected output but the type will be cast if necessary.

**fill_value**

[{var}, optional] Value used to fill in the masked values.

**keepdims**

[bool, optional] If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the array.

**Returns**

**ptp**

[ndarray.] A new array holding the result, unless `out` was specified, in which case a reference to `out` is returned.

## Examples

```
>>> x = np.ma.MaskedArray([[4, 9, 2, 10],
...                        [6, 9, 7, 12]])
```

```
>>> x.ptp(axis=1)
masked_array(data=[8, 6],
             mask=False,
       fill_value=999999)
```

```
>>> x.ptp(axis=0)
masked_array(data=[2, 0, 5, 2],
             mask=False,
       fill_value=999999)
```

```
>>> x.ptp()
10
```

This example shows that a negative value can be returned when the input is an array of signed integers.

```
>>> y = np.ma.MaskedArray([[1, 127],
...                        [0, 127],
...                        [-1, 127],
...                        [-2, 127]], dtype=np.int8)
>>> y.ptp(axis=1)
masked_array(data=[ 126,  127, -128, -127],
             mask=False,
       fill_value=999999,
            dtype=int8)
```

A work-around is to use the *view()* method to view the result as unsigned integers with the same bit width:

```
>>> y.ptp(axis=1).view(np.uint8)
masked_array(data=[126, 127, 128, 129],
```

(continues on next page)

(continued from previous page)

```
           mask=False,
     fill_value=999999,
          dtype=uint8)
```

method

`MaskedArray.`**`round`**(*self*, *decimals=0*, *out=None*)

Return each element rounded to the given number of decimals.

Refer to `numpy.around` for full documentation.

**See also:**

**`numpy.ndarray.around`**

corresponding function for ndarrays

**`numpy.around`**

equivalent function

method

`MaskedArray.`**`std`**(*self*, *axis=None*, *dtype=None*, *out=None*, *ddof=0*, *keepdims=<no value>*)

Returns the standard deviation of the array elements along given axis.

Masked entries are ignored.

Refer to `numpy.std` for full documentation.

**See also:**

**`numpy.ndarray.std`**

corresponding function for ndarrays

**`numpy.std`**

Equivalent function

method

`MaskedArray.`**`sum`**(*self*, *axis=None*, *dtype=None*, *out=None*, *keepdims=<no value>*)

Return the sum of the array elements over the given axis.

Masked elements are set to 0 internally.

Refer to `numpy.sum` for full documentation.

**See also:**

**`numpy.ndarray.sum`**

corresponding function for ndarrays

**`numpy.sum`**

equivalent function

### Examples

```
>>> x = np.ma.array([[1,2,3],[4,5,6],[7,8,9]], mask=[0] + [1,0]*4)
>>> x
masked_array(
  data=[[1, --, 3],
        [--, 5, --],
        [7, --, 9]],
  mask=[[False,  True, False],
        [ True, False,  True],
        [False,  True, False]],
  fill_value=999999)
>>> x.sum()
25
>>> x.sum(axis=1)
masked_array(data=[4, 5, 16],
             mask=[False, False, False],
       fill_value=999999)
>>> x.sum(axis=0)
masked_array(data=[8, 5, 12],
             mask=[False, False, False],
       fill_value=999999)
>>> print(type(x.sum(axis=0, dtype=np.int64)[0]))
<class 'numpy.int64'>
```

method

`MaskedArray`.**`trace`**(*offset=0*, *axis1=0*, *axis2=1*, *dtype=None*, *out=None*)

Return the sum along diagonals of the array.

Refer to `numpy.trace` for full documentation.

**See also:**

**`numpy.trace`**

equivalent function

method

`MaskedArray`.**`var`**(*self*, *axis=None*, *dtype=None*, *out=None*, *ddof=0*, *keepdims=<no value>*)

Compute the variance along the specified axis.

Returns the variance of the array elements, a measure of the spread of a distribution. The variance is computed for the flattened array by default, otherwise over the specified axis.

**Parameters**

**a**

[array_like] Array containing numbers whose variance is desired. If *a* is not an array, a conversion is attempted.

**axis**

[None or int or tuple of ints, optional] Axis or axes along which the variance is computed. The default is to compute the variance of the flattened array.

New in version 1.7.0.

If this is a tuple of ints, a variance is performed over multiple axes, instead of a single axis or all the axes as before.

**dtype**

[data-type, optional] Type to use in computing the variance. For arrays of integer type the default is *float64*; for arrays of float types it is the same as the array type.

**out**

[ndarray, optional] Alternate output array in which to place the result. It must have the same shape as the expected output, but the type is cast if necessary.

**ddof**

[int, optional] "Delta Degrees of Freedom": the divisor used in the calculation is `N - ddof`, where `N` represents the number of elements. By default *ddof* is zero.

**keepdims**

[bool, optional] If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the input array.

If the default value is passed, then *keepdims* will not be passed through to the `var` method of sub-classes of *ndarray*, however any non-default value will be. If the sub-class' method does not implement *keepdims* any exceptions will be raised.

**Returns**

**variance**

[ndarray, see dtype parameter above] If `out=None`, returns a new array containing the variance; otherwise, a reference to the output array is returned.

**See also:**

`std`, `mean`, `nanmean`, `nanstd`, `nanvar`, `ufuncs-output-type`

## Notes

The variance is the average of the squared deviations from the mean, i.e., `var = mean(abs(x - x.mean())**2)`.

The mean is normally calculated as `x.sum() / N`, where `N = len(x)`. If, however, *ddof* is specified, the divisor `N - ddof` is used instead. In standard statistical practice, `ddof=1` provides an unbiased estimator of the variance of a hypothetical infinite population. `ddof=0` provides a maximum likelihood estimate of the variance for normally distributed variables.

Note that for complex numbers, the absolute value is taken before squaring, so that the result is always real and nonnegative.

For floating-point input, the variance is computed using the same precision the input has. Depending on the input data, this can cause the results to be inaccurate, especially for *float32* (see example below). Specifying a higher-accuracy accumulator using the `dtype` keyword can alleviate this issue.

### Examples

```
>>> a = np.array([[1, 2], [3, 4]])
>>> np.var(a)
1.25
>>> np.var(a, axis=0)
array([1.,  1.])
>>> np.var(a, axis=1)
array([0.25,  0.25])
```

In single precision, var() can be inaccurate:

```
>>> a = np.zeros((2, 512*512), dtype=np.float32)
>>> a[0, :] = 1.0
>>> a[1, :] = 0.1
>>> np.var(a)
0.20250003
```

Computing the variance in float64 is more accurate:

```
>>> np.var(a, dtype=np.float64)
0.20249999932944759 # may vary
>>> ((1-0.55)**2 + (0.1-0.55)**2)/2
0.2025
```

## Arithmetic and comparison operations

### Comparison operators:

| | |
|---|---|
| *MaskedArray.__lt__*(self, value, /) | Return self<value. |
| *MaskedArray.__le__*(self, value, /) | Return self<=value. |
| *MaskedArray.__gt__*(self, value, /) | Return self>value. |
| *MaskedArray.__ge__*(self, value, /) | Return self>=value. |
| *MaskedArray.__eq__*(self, other) | Check whether other equals self elementwise. |
| *MaskedArray.__ne__*(self, other) | Check whether other does not equal self elementwise. |

method

`MaskedArray.`**`__lt__`**(*self*, *value*, /)
Return self<value.

method

`MaskedArray.`**`__le__`**(*self*, *value*, /)
Return self<=value.

method

`MaskedArray.`**`__gt__`**(*self*, *value*, /)
Return self>value.

method

`MaskedArray.`**`__ge__`**(*self*, *value*, /)
Return self>=value.

method

`MaskedArray.`**`__eq__`**`(`*self*, *other*`)`

Check whether other equals self elementwise.

When either of the elements is masked, the result is masked as well, but the underlying boolean data are still set, with self and other considered equal if both are masked, and unequal otherwise.

For structured arrays, all fields are combined, with masked values ignored. The result is masked if all fields were masked, with self and other considered equal only if both were fully masked.

method

`MaskedArray.`**`__ne__`**`(`*self*, *other*`)`

Check whether other does not equal self elementwise.

When either of the elements is masked, the result is masked as well, but the underlying boolean data are still set, with self and other considered equal if both are masked, and unequal otherwise.

For structured arrays, all fields are combined, with masked values ignored. The result is masked if all fields were masked, with self and other considered equal only if both were fully masked.

### Truth value of an array (`bool`):

| | |
|---|---|
| *`MaskedArray.__bool__`*(self, /) | self != 0 |

method

`MaskedArray.`**`__bool__`**`(`*self*, */*`)`

self != 0

### Arithmetic:

| | |
|---|---|
| *`MaskedArray.__abs__`*(self) | |
| *`MaskedArray.__add__`*(self, other) | Add self to other, and return a new masked array. |
| *`MaskedArray.__radd__`*(self, other) | Add other to self, and return a new masked array. |
| *`MaskedArray.__sub__`*(self, other) | Subtract other from self, and return a new masked array. |
| *`MaskedArray.__rsub__`*(self, other) | Subtract self from other, and return a new masked array. |
| *`MaskedArray.__mul__`*(self, other) | Multiply self by other, and return a new masked array. |
| *`MaskedArray.__rmul__`*(self, other) | Multiply other by self, and return a new masked array. |
| *`MaskedArray.__div__`*(self, other) | Divide other into self, and return a new masked array. |
| *`MaskedArray.__truediv__`*(self, other) | Divide other into self, and return a new masked array. |
| *`MaskedArray.__rtruediv__`*(self, other) | Divide self into other, and return a new masked array. |
| *`MaskedArray.__floordiv__`*(self, other) | Divide other into self, and return a new masked array. |
| *`MaskedArray.__rfloordiv__`*(self, other) | Divide self into other, and return a new masked array. |
| *`MaskedArray.__mod__`*(self, value, /) | Return self%value. |
| *`MaskedArray.__rmod__`*(self, value, /) | Return value%self. |
| *`MaskedArray.__divmod__`*(self, value, /) | Return divmod(self, value). |
| *`MaskedArray.__rdivmod__`*(self, value, /) | Return divmod(value, self). |
| *`MaskedArray.__pow__`*(self, other) | Raise self to the power other, masking the potential NaNs/Infs |
| *`MaskedArray.__rpow__`*(self, other) | Raise other to the power self, masking the potential NaNs/Infs |
| *`MaskedArray.__lshift__`*(self, value, /) | Return self«value. |
| *`MaskedArray.__rlshift__`*(self, value, /) | Return value«self. |
| *`MaskedArray.__rshift__`*(self, value, /) | Return self»value. |
| *`MaskedArray.__rrshift__`*(self, value, /) | Return value»self. |

Continued on next page

Table 60 – continued from previous page

| | |
|---|---|
| *MaskedArray.__and__*(self, value, /) | Return self&value. |
| *MaskedArray.__rand__*(self, value, /) | Return value&self. |
| *MaskedArray.__or__*(self, value, /) | Return self\|value. |
| *MaskedArray.__ror__*(self, value, /) | Return value\|self. |
| *MaskedArray.__xor__*(self, value, /) | Return self^value. |
| *MaskedArray.__rxor__*(self, value, /) | Return value^self. |

method

`MaskedArray.`**`__abs__`**(*self*)

method

`MaskedArray.`**`__add__`**(*self*, *other*)
Add self to other, and return a new masked array.

method

`MaskedArray.`**`__radd__`**(*self*, *other*)
Add other to self, and return a new masked array.

method

`MaskedArray.`**`__sub__`**(*self*, *other*)
Subtract other from self, and return a new masked array.

method

`MaskedArray.`**`__rsub__`**(*self*, *other*)
Subtract self from other, and return a new masked array.

method

`MaskedArray.`**`__mul__`**(*self*, *other*)
Multiply self by other, and return a new masked array.

method

`MaskedArray.`**`__rmul__`**(*self*, *other*)
Multiply other by self, and return a new masked array.

method

`MaskedArray.`**`__div__`**(*self*, *other*)
Divide other into self, and return a new masked array.

method

`MaskedArray.`**`__truediv__`**(*self*, *other*)
Divide other into self, and return a new masked array.

method

`MaskedArray.`**`__rtruediv__`**(*self*, *other*)
Divide self into other, and return a new masked array.

method

`MaskedArray.`**`__floordiv__`**(*self*, *other*)
Divide other into self, and return a new masked array.

method

`MaskedArray.`**`__rfloordiv__`**`(`*self*, *other*`)`
Divide self into other, and return a new masked array.

method

`MaskedArray.`**`__mod__`**`(`*self*, *value*, `/)`
Return self%value.

method

`MaskedArray.`**`__rmod__`**`(`*self*, *value*, `/)`
Return value%self.

method

`MaskedArray.`**`__divmod__`**`(`*self*, *value*, `/)`
Return divmod(self, value).

method

`MaskedArray.`**`__rdivmod__`**`(`*self*, *value*, `/)`
Return divmod(value, self).

method

`MaskedArray.`**`__pow__`**`(`*self*, *other*`)`
Raise self to the power other, masking the potential NaNs/Infs

method

`MaskedArray.`**`__rpow__`**`(`*self*, *other*`)`
Raise other to the power self, masking the potential NaNs/Infs

method

`MaskedArray.`**`__lshift__`**`(`*self*, *value*, `/)`
Return self«value.

method

`MaskedArray.`**`__rlshift__`**`(`*self*, *value*, `/)`
Return value«self.

method

`MaskedArray.`**`__rshift__`**`(`*self*, *value*, `/)`
Return self»value.

method

`MaskedArray.`**`__rrshift__`**`(`*self*, *value*, `/)`
Return value»self.

method

`MaskedArray.`**`__and__`**`(`*self*, *value*, `/)`
Return self&value.

method

`MaskedArray.`**`__rand__`**`(`*self*, *value*, `/)`
Return value&self.

method

`MaskedArray.`**`__or__`**`(`*self*, *value*, `/)`
Return self|value.

method

`MaskedArray.`**`__ror__`**`(`*self*, *value*, */*`)`
Return value|self.

method

`MaskedArray.`**`__xor__`**`(`*self*, *value*, */*`)`
Return self^value.

method

`MaskedArray.`**`__rxor__`**`(`*self*, *value*, */*`)`
Return value^self.

### Arithmetic, in-place:

| | |
|---|---|
| *`MaskedArray.__iadd__`*(self, other) | Add other to self in-place. |
| *`MaskedArray.__isub__`*(self, other) | Subtract other from self in-place. |
| *`MaskedArray.__imul__`*(self, other) | Multiply self by other in-place. |
| *`MaskedArray.__idiv__`*(self, other) | Divide self by other in-place. |
| *`MaskedArray.__itruediv__`*(self, other) | True divide self by other in-place. |
| *`MaskedArray.__ifloordiv__`*(self, other) | Floor divide self by other in-place. |
| *`MaskedArray.__imod__`*(self, value, /) | Return self%=value. |
| *`MaskedArray.__ipow__`*(self, other) | Raise self to the power other, in place. |
| *`MaskedArray.__ilshift__`*(self, value, /) | Return self«=value. |
| *`MaskedArray.__irshift__`*(self, value, /) | Return self»=value. |
| *`MaskedArray.__iand__`*(self, value, /) | Return self&=value. |
| *`MaskedArray.__ior__`*(self, value, /) | Return self\|=value. |
| *`MaskedArray.__ixor__`*(self, value, /) | Return self^=value. |

method

`MaskedArray.`**`__iadd__`**`(`*self*, *other*`)`
Add other to self in-place.

method

`MaskedArray.`**`__isub__`**`(`*self*, *other*`)`
Subtract other from self in-place.

method

`MaskedArray.`**`__imul__`**`(`*self*, *other*`)`
Multiply self by other in-place.

method

`MaskedArray.`**`__idiv__`**`(`*self*, *other*`)`
Divide self by other in-place.

method

`MaskedArray.`**`__itruediv__`**`(`*self*, *other*`)`
True divide self by other in-place.

method

`MaskedArray.`**`__ifloordiv__`**`(`*self*, *other*`)`
Floor divide self by other in-place.

method

`MaskedArray.`**`__imod__`**(*self*, *value*, */*)
    Return self%=value.

method

`MaskedArray.`**`__ipow__`**(*self*, *other*)
    Raise self to the power other, in place.

method

`MaskedArray.`**`__ilshift__`**(*self*, *value*, */*)
    Return self«=value.

method

`MaskedArray.`**`__irshift__`**(*self*, *value*, */*)
    Return self»=value.

method

`MaskedArray.`**`__iand__`**(*self*, *value*, */*)
    Return self&=value.

method

`MaskedArray.`**`__ior__`**(*self*, *value*, */*)
    Return self|=value.

method

`MaskedArray.`**`__ixor__`**(*self*, *value*, */*)
    Return self^=value.

## Representation

| | |
|---|---|
| `MaskedArray.__repr__`(self) | Literal string representation. |
| `MaskedArray.__str__`(self) | Return str(self). |
| `MaskedArray.ids`(self) | Return the addresses of the data and mask areas. |
| `MaskedArray.iscontiguous`(self) | Return a boolean indicating whether the data is contiguous. |

method

`MaskedArray.`**`__repr__`**(*self*)
    Literal string representation.

method

`MaskedArray.`**`__str__`**(*self*)
    Return str(self).

method

`MaskedArray.`**`ids`**(*self*)
    Return the addresses of the data and mask areas.

**Parameters**

**None**

### Examples

```
>>> x = np.ma.array([1, 2, 3], mask=[0, 1, 1])
>>> x.ids()
(166670640, 166659832) # may vary
```

If the array has no mask, the address of *nomask* is returned. This address is typically not close to the data in memory:

```
>>> x = np.ma.array([1, 2, 3])
>>> x.ids()
(166691080, 3083169284) # may vary
```

method

`MaskedArray.`**`iscontiguous`**`(`*self*`)`

Return a boolean indicating whether the data is contiguous.

**Parameters**

**None**

### Examples

```
>>> x = np.ma.array([1, 2, 3])
>>> x.iscontiguous()
True
```

*iscontiguous* returns one of the flags of the masked array:

```
>>> x.flags
  C_CONTIGUOUS : True
  F_CONTIGUOUS : True
  OWNDATA : False
  WRITEABLE : True
  ALIGNED : True
  WRITEBACKIFCOPY : False
  UPDATEIFCOPY : False
```

## Special methods

For standard library functions:

| | |
|---|---|
| *MaskedArray.__copy__*() | Used if `copy.copy` is called on an array. |
| *MaskedArray.__deepcopy__*(self[, memo]) | Used if `copy.deepcopy` is called on an array. |
| *MaskedArray.__getstate__*(self) | Return the internal state of the masked array, for pickling purposes. |
| *MaskedArray.__reduce__*(self) | Return a 3-tuple for pickling a MaskedArray. |
| *MaskedArray.__setstate__*(self, state) | Restore the internal state of the masked array, for pickling purposes. |

method

`MaskedArray.`**`__copy__`**`()`

Used if `copy.copy` is called on an array. Returns a copy of the array.

Equivalent to `a.copy(order='K')`.

method

`MaskedArray.`**`__deepcopy__`**`(`*self*, *memo=None*`)`

Used if `copy.deepcopy` is called on an array.

method

`MaskedArray.`**`__getstate__`**`(`*self*`)`

Return the internal state of the masked array, for pickling purposes.

method

`MaskedArray.`**`__reduce__`**`(`*self*`)`

Return a 3-tuple for pickling a MaskedArray.

method

`MaskedArray.`**`__setstate__`**`(`*self*, *state*`)`

Restore the internal state of the masked array, for pickling purposes. `state` is typically the output of the `__getstate__` output, and is a 5-tuple:

- class name
- a tuple giving the shape of the data
- a typecode for the data
- a binary string for the data
- a binary string for the mask.

Basic customization:

| | |
|---|---|
| `MaskedArray.__new__`(cls[, data, mask, …]) | Create a new masked array from scratch. |
| `MaskedArray.__array__`() | Returns either a new reference to self if dtype is not given or a new array of provided data type if dtype is different from the current dtype of the array. |
| `MaskedArray.__array_wrap__`(self, obj[, context]) | Special hook for ufuncs. |

method

**`static`** `MaskedArray.`**`__new__`**`(`*cls*, *data=None*, *mask=False*, *dtype=None*, *copy=False*, *subok=True*, *ndmin=0*, *fill_value=None*, *keep_mask=True*, *hard_mask=None*, *shrink=True*, *order=None*, ***options*`)`

Create a new masked array from scratch.

### Notes

A masked array can also be created by taking a .view(MaskedArray).

method

`MaskedArray.`**`__array__`**`()`

Returns either a new reference to self if dtype is not given or a new array of provided data type if dtype is different from the current dtype of the array.

method

`MaskedArray.`**`__array_wrap__`**`(`*self*, *obj*, *context=None*`)`

Special hook for ufuncs.

Wraps the numpy array and sets the mask according to context.

Container customization: (see *Indexing*)

| | |
|---|---|
| `MaskedArray.__len__`(self, /) | Return len(self). |
| `MaskedArray.__getitem__`(self, indx) | x.__getitem__(y) <==> x[y] |
| `MaskedArray.__setitem__`(self, indx, value) | x.__setitem__(i, y) <==> x[i]=y |
| `MaskedArray.__delitem__`(self, key, /) | Delete self[key]. |
| `MaskedArray.__contains__`(self, key, /) | Return key in self. |

method

`MaskedArray.`**`__len__`**`(`*self*, */*`)`

Return len(self).

method

`MaskedArray.`**`__getitem__`**`(`*self*, *indx*`)`

x.__getitem__(y) <==> x[y]

Return the item described by i, as a masked array.

method

`MaskedArray.`**`__setitem__`**`(`*self*, *indx*, *value*`)`

x.__setitem__(i, y) <==> x[i]=y

Set item described by index. If value is masked, masks those locations.

method

`MaskedArray.`**`__delitem__`**`(`*self*, *key*, */*`)`

Delete self[key].

method

`MaskedArray.`**`__contains__`**`(`*self*, *key*, */*`)`

Return key in self.

### Specific methods

#### Handling the mask

The following methods can be used to access information about the mask or to manipulate the mask.

| | |
|---|---|
| `MaskedArray.__setmask__`(self, mask[, copy]) | Set the mask. |
| `MaskedArray.harden_mask`(self) | Force the mask to hard. |
| `MaskedArray.soften_mask`(self) | Force the mask to soft. |
| `MaskedArray.unshare_mask`(self) | Copy the mask and set the sharedmask flag to False. |
| `MaskedArray.shrink_mask`(self) | Reduce a mask to nomask when possible. |

method

`MaskedArray.`**`__setmask__`**(*self*, *mask*, *copy=False*)

Set the mask.

method

`MaskedArray.`**`harden_mask`**(*self*)

Force the mask to hard.

Whether the mask of a masked array is hard or soft is determined by its `hardmask` property. `harden_mask` sets `hardmask` to True.

**See also:**

`hardmask`

method

`MaskedArray.`**`soften_mask`**(*self*)

Force the mask to soft.

Whether the mask of a masked array is hard or soft is determined by its `hardmask` property. `soften_mask` sets `hardmask` to False.

**See also:**

`hardmask`

method

`MaskedArray.`**`unshare_mask`**(*self*)

Copy the mask and set the sharedmask flag to False.

Whether the mask is shared between masked arrays can be seen from the `sharedmask` property. `unshare_mask` ensures the mask is not shared. A copy of the mask is only made if it was shared.

**See also:**

`sharedmask`

method

`MaskedArray.`**`shrink_mask`**(*self*)

Reduce a mask to nomask when possible.

**Parameters**

**None**

**Returns**

**None**

### Examples

```
>>> x = np.ma.array([[1,2 ], [3, 4]], mask=[0]*4)
>>> x.mask
array([[False, False],
       [False, False]])
>>> x.shrink_mask()
masked_array(
  data=[[1, 2],
        [3, 4]],
  mask=False,
  fill_value=999999)
>>> x.mask
False
```

## Handling the *fill_value*

| | |
|---|---|
| *MaskedArray.get_fill_value*(self) | The filling value of the masked array is a scalar. |
| *MaskedArray.set_fill_value*(self[, value]) | |

method

`MaskedArray.`**`get_fill_value`**(*self*)

The filling value of the masked array is a scalar. When setting, None will set to a default based on the data type.

### Examples

```
>>> for dt in [np.int32, np.int64, np.float64, np.complex128]:
...     np.ma.array([0, 1], dtype=dt).get_fill_value()
...
999999
999999
1e+20
(1e+20+0j)
```

```
>>> x = np.ma.array([0, 1.], fill_value=-np.inf)
>>> x.fill_value
-inf
>>> x.fill_value = np.pi
>>> x.fill_value
3.1415926535897931 # may vary
```

Reset to default:

```
>>> x.fill_value = None
>>> x.fill_value
1e+20
```

method

`MaskedArray.`**`set_fill_value`**(*self*, *value=None*)

### Counting the missing elements

| | |
|---|---|
| *MaskedArray.count*(self[, axis, keepdims]) | Count the non-masked elements of the array along the given axis. |

method

`MaskedArray.`**`count`** (*self*, *axis=None*, *keepdims=<no value>*)

Count the non-masked elements of the array along the given axis.

**Parameters**

**axis**

[None or int or tuple of ints, optional] Axis or axes along which the count is performed. The default, None, performs the count over all the dimensions of the input array. *axis* may be negative, in which case it counts from the last to the first axis.

New in version 1.10.0.

If this is a tuple of ints, the count is performed on multiple axes, instead of a single axis or all the axes as before.

**keepdims**

[bool, optional] If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the array.

**Returns**

**result**

[ndarray or scalar] An array with the same shape as the input array, with the specified axis removed. If the array is a 0-d array, or if *axis* is None, a scalar is returned.

**See also:**

***count_masked***

Count masked elements in array or along a given axis.

#### Examples

```
>>> import numpy.ma as ma
>>> a = ma.arange(6).reshape((2, 3))
>>> a[1, :] = ma.masked
>>> a
masked_array(
  data=[[0, 1, 2],
        [--, --, --]],
  mask=[[False, False, False],
        [ True,  True,  True]],
  fill_value=999999)
>>> a.count()
3
```

When the *axis* keyword is specified an array of appropriate size is returned.

```
>>> a.count(axis=0)
array([1, 1, 1])
>>> a.count(axis=1)
array([3, 0])
```

## 1.7.7 Masked array operations

### Constants

| | |
|---|---|
| `ma.MaskType` | alias of `numpy.bool_` |

`numpy.ma.`**`MaskType`**
alias of `numpy.bool_`

### Creation

### From existing data

| | |
|---|---|
| `ma.masked_array` | alias of `numpy.ma.core.MaskedArray` |
| `ma.array`(data[, dtype, copy, order, mask, …]) | An array class with possibly masked values. |
| `ma.copy`(self, *args, **params) a.copy(order=) | Return a copy of the array. |
| `ma.frombuffer`(buffer[, dtype, count, offset]) | Interpret a buffer as a 1-dimensional array. |
| `ma.fromfunction`(function, shape, **dtype) | Construct an array by executing a function over each coordinate. |
| `ma.MaskedArray.copy`([order]) | Return a copy of the array. |

`numpy.ma.`**`copy`**(*self, *args, **params) a.copy(order='C')*) **`= <numpy.ma.core._frommethod object>`**

Return a copy of the array.

**Parameters**

**order**

[{'C', 'F', 'A', 'K'}, optional] Controls the memory layout of the copy. 'C' means C-order, 'F' means F-order, 'A' means 'F' if *a* is Fortran contiguous, 'C' otherwise. 'K' means match the layout of *a* as closely as possible. (Note that this function and `numpy.copy` are very similar, but have different default values for their order= arguments.)

### Examples

```
>>> x = np.array([[1,2,3],[4,5,6]], order='F')
```

```
>>> y = x.copy()
```

```
>>> x.fill(0)
>>> x
array([[0, 0, 0],
       [0, 0, 0]])
```

```
>>> y
array([[1, 2, 3],
       [4, 5, 6]])
```

```
>>> y.flags['C_CONTIGUOUS']
True
```

`numpy.ma.`**`frombuffer`**(*buffer*, *dtype=float*, *count=-1*, *offset=0*) **`= <numpy.ma.core._convert2ma object>`**

Interpret a buffer as a 1-dimensional array.

**Parameters**

**buffer**

[buffer_like] An object that exposes the buffer interface.

**dtype**

[data-type, optional] Data-type of the returned array; default: float.

**count**

[int, optional] Number of items to read. `-1` means all data in the buffer.

**offset**

[int, optional] Start reading the buffer from this offset (in bytes); default: 0.

## Notes

If the buffer has data that is not in machine byte-order, this should be specified as part of the data-type, e.g.:

```
>>> dt = np.dtype(int)
>>> dt = dt.newbyteorder('>')
>>> np.frombuffer(buf, dtype=dt)
```

The data of the resulting array will not be byteswapped, but will be interpreted correctly.

## Examples

```
>>> s = b'hello world'
>>> np.frombuffer(s, dtype='S1', count=5, offset=6)
array([b'w', b'o', b'r', b'l', b'd'], dtype='|S1')
```

```
>>> np.frombuffer(b'\x01\x02', dtype=np.uint8)
array([1, 2], dtype=uint8)
>>> np.frombuffer(b'\x01\x02\x03\x04\x05', dtype=np.uint8, count=3)
array([1, 2, 3], dtype=uint8)
```

`numpy.ma.`**`fromfunction`**(*function*, *shape*, ***dtype*) **`= <numpy.ma.core._convert2ma object>`**

Construct an array by executing a function over each coordinate.

The resulting array therefore has a value `fn(x, y, z)` at coordinate `(x, y, z)`.

**Parameters**

**function**

[callable] The function is called with N parameters, where N is the rank of *shape*. Each parameter represents the coordinates of the array varying along a specific axis. For example, if *shape* were `(2, 2)`, then the parameters would be `array([[0, 0], [1, 1]])` and `array([[0, 1], [0, 1]])`

**shape**

[(N,) tuple of ints] Shape of the output array, which also determines the shape of the coordinate arrays passed to *function*.

**dtype**

[data-type, optional] Data-type of the coordinate arrays passed to *function*. By default, *dtype* is float.

**Returns**

**fromfunction**

[any] The result of the call to *function* is passed back directly. Therefore the shape of *fromfunction* is completely determined by *function*. If *function* returns a scalar value, the shape of *fromfunction* would not match the *shape* parameter.

**See also:**

*indices*, `meshgrid`

## Notes

Keywords other than *dtype* are passed to *function*.

## Examples

```
>>> np.fromfunction(lambda i, j: i == j, (3, 3), dtype=int)
array([[ True, False, False],
       [False,  True, False],
       [False, False,  True]])
```

```
>>> np.fromfunction(lambda i, j: i + j, (3, 3), dtype=int)
array([[0, 1, 2],
       [1, 2, 3],
       [2, 3, 4]])
```

### Ones and zeros

| | |
|---|---|
| `ma.empty`(shape[, dtype, order]) | Return a new array of given shape and type, without initializing entries. |
| `ma.empty_like`(prototype[, dtype, order, …]) | Return a new array with the same shape and type as a given array. |
| `ma.masked_all`(shape[, dtype]) | Empty masked array with all elements masked. |
| `ma.masked_all_like`(arr) | Empty masked array with the properties of an existing array. |
| `ma.ones`(shape[, dtype, order]) | Return a new array of given shape and type, filled with ones. |
| `ma.zeros`(shape[, dtype, order]) | Return a new array of given shape and type, filled with zeros. |

`numpy.ma.`**`empty`**(*shape*, *dtype=float*, *order='C'*) **`= <numpy.ma.core._convert2ma object>`**

Return a new array of given shape and type, without initializing entries.

**Parameters**

**shape**

[int or tuple of int] Shape of the empty array, e.g., `(2, 3)` or `2`.

**dtype**

[data-type, optional] Desired output data-type for the array, e.g, `numpy.int8`. Default is `numpy.float64`.

**order**

[{'C', 'F'}, optional, default: 'C'] Whether to store multi-dimensional data in row-major (C-style) or column-major (Fortran-style) order in memory.

**Returns**

**out**

[ndarray] Array of uninitialized (arbitrary) data of the given shape, dtype, and order. Object arrays will be initialized to None.

**See also:**

***`empty_like`***

Return an empty array with shape and type of input.

***`ones`***

Return a new array setting values to one.

***`zeros`***

Return a new array setting values to zero.

**`full`**

Return a new array of given shape filled with value.

## Notes

*empty*, unlike *zeros*, does not set the array values to zero, and may therefore be marginally faster. On the other hand, it requires the user to manually set all the values in the array, and should be used with caution.

## Examples

```
>>> np.empty([2, 2])
array([[ -9.74499359e+001,   6.69583040e-309],
       [  2.13182611e-314,   3.06959433e-309]])         #uninitialized
```

```
>>> np.empty([2, 2], dtype=int)
array([[-1073741821, -1067949133],
       [  496041986,    19249760]])                     #uninitialized
```

`numpy.ma.`**`empty_like`**(*prototype*, *dtype=None*, *order='K'*, *subok=True*, *shape=None*) **`= <numpy.ma.core._convert2ma object>`**

Return a new array with the same shape and type as a given array.

**Parameters**

**prototype**

[array_like] The shape and data-type of *prototype* define these same attributes of the returned array.

**dtype**

[data-type, optional] Overrides the data type of the result.

New in version 1.6.0.

**order**

[{'C', 'F', 'A', or 'K'}, optional] Overrides the memory layout of the result. 'C' means C-order, 'F' means F-order, 'A' means 'F' if `prototype` is Fortran contiguous, 'C' otherwise. 'K' means match the layout of `prototype` as closely as possible.

New in version 1.6.0.

**subok**

[bool, optional.] If True, then the newly created array will use the sub-class type of 'a', otherwise it will be a base-class array. Defaults to True.

**shape**

[int or sequence of ints, optional.] Overrides the shape of the result. If order='K' and the number of dimensions is unchanged, will try to keep order, otherwise, order='C' is implied.

New in version 1.17.0.

**Returns**

**out**

[ndarray] Array of uninitialized (arbitrary) data with the same shape and type as *prototype*.

**See also:**

**`ones_like`**

Return an array of ones with shape and type of input.

**`zeros_like`**

Return an array of zeros with shape and type of input.

**`full_like`**

Return a new array with shape of input filled with value.

***`empty`***

Return a new uninitialized array.

### Notes

This function does *not* initialize the returned array; to do that use *zeros_like* or *ones_like* instead. It may be marginally faster than the functions that do set the array values.

### Examples

```
>>> a = ([1,2,3], [4,5,6])                         # a is array-like
>>> np.empty_like(a)
array([[-1073741821, -1073741821,           3],    # uninitialized
       [          0,           0, -1073741821]])
>>> a = np.array([[1., 2., 3.],[4.,5.,6.]])
>>> np.empty_like(a)
array([[ -2.00000715e+000,   1.48219694e-323,  -2.00000572e+000], # uninitialized
       [  4.38791518e-305,  -2.00000715e+000,   4.17269252e-309]])
```

`numpy.ma.`**`masked_all`**(*shape*, *dtype=<class 'float'>*)

Empty masked array with all elements masked.

Return an empty masked array of the given shape and dtype, where all the data are masked.

**Parameters**

**shape**

[tuple] Shape of the required MaskedArray.

**dtype**

[dtype, optional] Data type of the output.

**Returns**

**a**

[MaskedArray] A masked array with all data masked.

**See also:**

***`masked_all_like`***

Empty masked array modelled on an existing array.

### Examples

```
>>> import numpy.ma as ma
>>> ma.masked_all((3, 3))
masked_array(
  data=[[--, --, --],
        [--, --, --],
        [--, --, --]],
  mask=[[ True,  True,  True],
        [ True,  True,  True],
        [ True,  True,  True]],
  fill_value=1e+20,
  dtype=float64)
```

The *dtype* parameter defines the underlying data type.

```
>>> a = ma.masked_all((3, 3))
>>> a.dtype
dtype('float64')
>>> a = ma.masked_all((3, 3), dtype=np.int32)
>>> a.dtype
dtype('int32')
```

`numpy.ma.`**`masked_all_like`**`(`*arr*`)`

Empty masked array with the properties of an existing array.

Return an empty masked array of the same shape and dtype as the array *arr*, where all the data are masked.

**Parameters**

**arr**

[ndarray] An array describing the shape and dtype of the required MaskedArray.

**Returns**

**a**

[MaskedArray] A masked array with all data masked.

**Raises**

**AttributeError**

If *arr* doesn't have a shape attribute (i.e. not an ndarray)

**See also:**

***`masked_all`***

Empty masked array with all elements masked.

### Examples

```
>>> import numpy.ma as ma
>>> arr = np.zeros((2, 3), dtype=np.float32)
>>> arr
array([[0., 0., 0.],
       [0., 0., 0.]], dtype=float32)
>>> ma.masked_all_like(arr)
masked_array(
  data=[[--, --, --],
        [--, --, --]],
  mask=[[ True,  True,  True],
        [ True,  True,  True]],
  fill_value=1e+20,
  dtype=float32)
```

The dtype of the masked array matches the dtype of *arr*.

```
>>> arr.dtype
dtype('float32')
>>> ma.masked_all_like(arr).dtype
dtype('float32')
```

`numpy.ma.`**`ones`**(*shape*, *dtype=None*, *order='C'*) **`= <numpy.ma.core._convert2ma object>`**

Return a new array of given shape and type, filled with ones.

**Parameters**

**shape**

[int or sequence of ints] Shape of the new array, e.g., `(2, 3)` or `2`.

**dtype**

[data-type, optional] The desired data-type for the array, e.g., `numpy.int8`. Default is `numpy.float64`.

**order**

[{'C', 'F'}, optional, default: C] Whether to store multi-dimensional data in row-major (C-style) or column-major (Fortran-style) order in memory.

**Returns**

**out**

[ndarray] Array of ones with the given shape, dtype, and order.

**See also:**

**`ones_like`**

Return an array of ones with shape and type of input.

*`empty`*

Return a new uninitialized array.

*`zeros`*

Return a new array setting values to zero.

**`full`**

Return a new array of given shape filled with value.

### Examples

```
>>> np.ones(5)
array([1., 1., 1., 1., 1.])
```

```
>>> np.ones((5,), dtype=int)
array([1, 1, 1, 1, 1])
```

```
>>> np.ones((2, 1))
array([[1.],
       [1.]])
```

```
>>> s = (2,2)
>>> np.ones(s)
array([[1.,  1.],
       [1.,  1.]])
```

`numpy.ma.`**`zeros`**(*shape*, *dtype=float*, *order='C'*) **`= <numpy.ma.core._convert2ma object>`**

Return a new array of given shape and type, filled with zeros.

**Parameters**

**shape**

[int or tuple of ints] Shape of the new array, e.g., `(2, 3)` or `2`.

**dtype**

[data-type, optional] The desired data-type for the array, e.g., `numpy.int8`. Default is `numpy.float64`.

**order**

[{'C', 'F'}, optional, default: 'C'] Whether to store multi-dimensional data in row-major (C-style) or column-major (Fortran-style) order in memory.

**Returns**

**out**

[ndarray] Array of zeros with the given shape, dtype, and order.

**See also:**

**`zeros_like`**

Return an array of zeros with shape and type of input.

***`empty`***

Return a new uninitialized array.

***`ones`***

Return a new array setting values to one.

**full**

Return a new array of given shape filled with value.

### Examples

```
>>> np.zeros(5)
array([ 0.,  0.,  0.,  0.,  0.])
```

```
>>> np.zeros((5,), dtype=int)
array([0, 0, 0, 0, 0])
```

```
>>> np.zeros((2, 1))
array([[ 0.],
       [ 0.]])
```

```
>>> s = (2,2)
>>> np.zeros(s)
array([[ 0.,  0.],
       [ 0.,  0.]])
```

```
>>> np.zeros((2,), dtype=[('x', 'i4'), ('y', 'i4')]) # custom dtype
array([(0, 0), (0, 0)],
      dtype=[('x', '<i4'), ('y', '<i4')])
```

---

## Inspecting the array

| | |
|---|---|
| `ma.all`(self[, axis, out, keepdims]) | Returns True if all elements evaluate to True. |
| `ma.any`(self[, axis, out, keepdims]) | Returns True if any of the elements of *a* evaluate to True. |
| `ma.count`(self[, axis, keepdims]) | Count the non-masked elements of the array along the given axis. |
| `ma.count_masked`(arr[, axis]) | Count the number of masked elements along the given axis. |
| `ma.getmask`(a) | Return the mask of a masked array, or nomask. |
| `ma.getmaskarray`(arr) | Return the mask of a masked array, or full boolean array of False. |
| `ma.getdata`(a[, subok]) | Return the data of a masked array as an ndarray. |
| `ma.nonzero`(self) | Return the indices of unmasked elements that are not zero. |
| `ma.shape`(obj) | Return the shape of an array. |
| `ma.size`(obj[, axis]) | Return the number of elements along a given axis. |
| `ma.is_masked`(x) | Determine whether input has masked values. |
| `ma.is_mask`(m) | Return True if m is a valid, standard mask. |
| `ma.MaskedArray.all`(self[, axis, out, keepdims]) | Returns True if all elements evaluate to True. |
| `ma.MaskedArray.any`(self[, axis, out, keepdims]) | Returns True if any of the elements of *a* evaluate to True. |
| `ma.MaskedArray.count`(self[, axis, keepdims]) | Count the non-masked elements of the array along the given axis. |
| `ma.MaskedArray.nonzero`(self) | Return the indices of unmasked elements that are not zero. |

Continued on next page

Table 72 – continued from previous page

| | |
|---|---|
| `ma.shape`(obj) | Return the shape of an array. |
| `ma.size`(obj[, axis]) | Return the number of elements along a given axis. |

`numpy.ma.`**`all`**(*self*, *axis=None*, *out=None*, *keepdims=<no value>*) **`= <numpy.ma.core._frommethod object>`**

Returns True if all elements evaluate to True.

The output array is masked where all the values along the given axis are masked: if the output would have been a scalar and that all the values are masked, then the output is `masked`.

Refer to `numpy.all` for full documentation.

**See also:**

**`numpy.ndarray.all`**

corresponding function for ndarrays

**`numpy.all`**

equivalent function

## Examples

```
>>> np.ma.array([1,2,3]).all()
True
>>> a = np.ma.array([1,2,3], mask=True)
>>> (a.all() is np.ma.masked)
True
```

`numpy.ma.`**`any`**(*self*, *axis=None*, *out=None*, *keepdims=<no value>*) **`= <numpy.ma.core._frommethod object>`**

Returns True if any of the elements of *a* evaluate to True.

Masked values are considered as False during computation.

Refer to `numpy.any` for full documentation.

**See also:**

**`numpy.ndarray.any`**

corresponding function for ndarrays

**`numpy.any`**

equivalent function

`numpy.ma.`**`count`**(*self*, *axis=None*, *keepdims=<no value>*) **`= <numpy.ma.core._frommethod object>`**

Count the non-masked elements of the array along the given axis.

**Parameters**

**axis**

[None or int or tuple of ints, optional] Axis or axes along which the count is performed. The default, None, performs the count over all the dimensions of the input array. *axis* may be negative, in which case it counts from the last to the first axis.

New in version 1.10.0.

If this is a tuple of ints, the count is performed on multiple axes, instead of a single axis or all the axes as before.

**keepdims**

[bool, optional] If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the array.

**Returns**

**result**

[ndarray or scalar] An array with the same shape as the input array, with the specified axis removed. If the array is a 0-d array, or if *axis* is None, a scalar is returned.

**See also:**

`count_masked`

Count masked elements in array or along a given axis.

## Examples

```
>>> import numpy.ma as ma
>>> a = ma.arange(6).reshape((2, 3))
>>> a[1, :] = ma.masked
>>> a
masked_array(
  data=[[0, 1, 2],
        [--, --, --]],
  mask=[[False, False, False],
        [ True,  True,  True]],
  fill_value=999999)
>>> a.count()
3
```

When the *axis* keyword is specified an array of appropriate size is returned.

```
>>> a.count(axis=0)
array([1, 1, 1])
>>> a.count(axis=1)
array([3, 0])
```

`numpy.ma.`**`count_masked`**(*arr*, *axis=None*)

Count the number of masked elements along the given axis.

**Parameters**

**arr**

[array_like] An array with (possibly) masked elements.

**axis**

[int, optional] Axis along which to count. If None (default), a flattened version of the array is used.

**Returns**

**count**

[int, ndarray] The total number of masked elements (axis=None) or the number of masked elements along each slice of the given axis.

**See also:**

***MaskedArray.count***

Count non-masked elements.

### Examples

```
>>> import numpy.ma as ma
>>> a = np.arange(9).reshape((3,3))
>>> a = ma.array(a)
>>> a[1, 0] = ma.masked
>>> a[1, 2] = ma.masked
>>> a[2, 1] = ma.masked
>>> a
masked_array(
  data=[[0, 1, 2],
        [--, 4, --],
        [6, --, 8]],
  mask=[[False, False, False],
        [ True, False,  True],
        [False,  True, False]],
  fill_value=999999)
>>> ma.count_masked(a)
3
```

When the *axis* keyword is used an array is returned.

```
>>> ma.count_masked(a, axis=0)
array([1, 1, 1])
>>> ma.count_masked(a, axis=1)
array([0, 2, 1])
```

`numpy.ma.`**`getmask`**`(`*a*`)`

Return the mask of a masked array, or nomask.

Return the mask of *a* as an ndarray if *a* is a *MaskedArray* and the mask is not *nomask*, else return *nomask*. To guarantee a full array of booleans of the same shape as a, use *getmaskarray*.

**Parameters**

**a**

[array_like] Input *MaskedArray* for which the mask is required.

**See also:**

***getdata***

Return the data of a masked array as an ndarray.

**`getmaskarray`**

Return the mask of a masked array, or full array of False.

## Examples

```
>>> import numpy.ma as ma
>>> a = ma.masked_equal([[1,2],[3,4]], 2)
>>> a
masked_array(
  data=[[1, --],
        [3, 4]],
  mask=[[False,  True],
        [False, False]],
  fill_value=2)
>>> ma.getmask(a)
array([[False,  True],
       [False, False]])
```

Equivalently use the `MaskedArray` *mask* attribute.

```
>>> a.mask
array([[False,  True],
       [False, False]])
```

Result when mask == `nomask`

```
>>> b = ma.masked_array([[1,2],[3,4]])
>>> b
masked_array(
  data=[[1, 2],
        [3, 4]],
  mask=False,
  fill_value=999999)
>>> ma.nomask
False
>>> ma.getmask(b) == ma.nomask
True
>>> b.mask == ma.nomask
True
```

`numpy.ma.`**`getmaskarray`**`(`*arr*`)`

Return the mask of a masked array, or full boolean array of False.

Return the mask of *arr* as an ndarray if *arr* is a `MaskedArray` and the mask is not `nomask`, else return a full boolean array of False of the same shape as *arr*.

**Parameters**

**arr**

[array_like] Input `MaskedArray` for which the mask is required.

**See also:**

**`getmask`**

Return the mask of a masked array, or nomask.

***getdata***

Return the data of a masked array as an ndarray.

### Examples

```
>>> import numpy.ma as ma
>>> a = ma.masked_equal([[1,2],[3,4]], 2)
>>> a
masked_array(
  data=[[1, --],
        [3, 4]],
  mask=[[False,  True],
        [False, False]],
  fill_value=2)
>>> ma.getmaskarray(a)
array([[False,  True],
       [False, False]])
```

Result when mask == `nomask`

```
>>> b = ma.masked_array([[1,2],[3,4]])
>>> b
masked_array(
  data=[[1, 2],
        [3, 4]],
  mask=False,
  fill_value=999999)
>>> ma.getmaskarray(b)
array([[False, False],
       [False, False]])
```

`numpy.ma.`**`getdata`**(*a*, *subok=True*)

Return the data of a masked array as an ndarray.

Return the data of *a* (if any) as an ndarray if *a* is a `MaskedArray`, else return *a* as a ndarray or subclass (depending on *subok*) if not.

**Parameters**

**a**

[array_like] Input `MaskedArray`, alternatively a ndarray or a subclass thereof.

**subok**

[bool] Whether to force the output to be a *pure* ndarray (False) or to return a subclass of ndarray if appropriate (True, default).

**See also:**

***getmask***

Return the mask of a masked array, or nomask.

***getmaskarray***

Return the mask of a masked array, or full array of False.

### Examples

```
>>> import numpy.ma as ma
>>> a = ma.masked_equal([[1,2],[3,4]], 2)
>>> a
masked_array(
  data=[[1, --],
        [3, 4]],
  mask=[[False,  True],
        [False, False]],
  fill_value=2)
>>> ma.getdata(a)
array([[1, 2],
       [3, 4]])
```

Equivalently use the `MaskedArray` *data* attribute.

```
>>> a.data
array([[1, 2],
       [3, 4]])
```

`numpy.ma.`**`nonzero`**`(`*self*`)` **`= <numpy.ma.core._frommethod object>`**

Return the indices of unmasked elements that are not zero.

Returns a tuple of arrays, one for each dimension, containing the indices of the non-zero elements in that dimension. The corresponding non-zero values can be obtained with:

```
a[a.nonzero()]
```

To group the indices by element, rather than dimension, use instead:

```
np.transpose(a.nonzero())
```

The result of this is always a 2d array, with a row for each non-zero element.

**Parameters**

**None**

**Returns**

**tuple_of_arrays**

[tuple] Indices of elements that are non-zero.

**See also:**

***`numpy.nonzero`***

Function operating on ndarrays.

**`flatnonzero`**

Return indices that are non-zero in the flattened version of the input array.

***`numpy.ndarray.nonzero`***

Equivalent ndarray method.

**`count_nonzero`**

Counts the number of non-zero elements in the input array.

## Examples

```
>>> import numpy.ma as ma
>>> x = ma.array(np.eye(3))
>>> x
masked_array(
  data=[[1., 0., 0.],
        [0., 1., 0.],
        [0., 0., 1.]],
  mask=False,
  fill_value=1e+20)
>>> x.nonzero()
(array([0, 1, 2]), array([0, 1, 2]))
```

Masked elements are ignored.

```
>>> x[1, 1] = ma.masked
>>> x
masked_array(
  data=[[1.0, 0.0, 0.0],
        [0.0, --, 0.0],
        [0.0, 0.0, 1.0]],
  mask=[[False, False, False],
        [False,  True, False],
        [False, False, False]],
  fill_value=1e+20)
>>> x.nonzero()
(array([0, 2]), array([0, 2]))
```

Indices can also be grouped by element.

```
>>> np.transpose(x.nonzero())
array([[0, 0],
       [2, 2]])
```

A common use for `nonzero` is to find the indices of an array, where a condition is True. Given an array *a*, the condition *a* > 3 is a boolean array and since False is interpreted as 0, ma.nonzero(a > 3) yields the indices of the *a* where the condition is true.

```
>>> a = ma.array([[1,2,3],[4,5,6],[7,8,9]])
>>> a > 3
masked_array(
  data=[[False, False, False],
        [ True,  True,  True],
        [ True,  True,  True]],
  mask=False,
  fill_value=True)
>>> ma.nonzero(a > 3)
(array([1, 1, 1, 2, 2, 2]), array([0, 1, 2, 0, 1, 2]))
```

The `nonzero` method of the condition array can also be called.

```
>>> (a > 3).nonzero()
(array([1, 1, 1, 2, 2, 2]), array([0, 1, 2, 0, 1, 2]))
```

`numpy.ma.`**`shape`**`(`*obj*`)`

Return the shape of an array.

**Parameters**

**a**

[array_like] Input array.

**Returns**

**shape**

[tuple of ints] The elements of the shape tuple give the lengths of the corresponding array dimensions.

**See also:**

`alen`

**`ndarray.shape`**

Equivalent array method.

### Examples

```
>>> np.shape(np.eye(3))
(3, 3)
>>> np.shape([[1, 2]])
(1, 2)
>>> np.shape([0])
(1,)
>>> np.shape(0)
()
```

```
>>> a = np.array([(1, 2), (3, 4)], dtype=[('x', 'i4'), ('y', 'i4')])
>>> np.shape(a)
(2,)
>>> a.shape
(2,)
```

`numpy.ma.`**`size`**`(`*obj, axis=None*`)`

Return the number of elements along a given axis.

**Parameters**

**a**

[array_like] Input data.

**axis**

[int, optional] Axis along which the elements are counted. By default, give the total number of elements.

**Returns**

**element_count**

[int] Number of elements along the specified axis.

**See also:**

***shape***

dimensions of array

**`ndarray.shape`**

dimensions of array

**`ndarray.size`**

number of elements in array

## Examples

```
>>> a = np.array([[1,2,3],[4,5,6]])
>>> np.size(a)
6
>>> np.size(a,1)
3
>>> np.size(a,0)
2
```

`numpy.ma.`**`is_masked`**`(`*x*`)`

Determine whether input has masked values.

Accepts any object as input, but always returns False unless the input is a MaskedArray containing masked values.

**Parameters**

**x**

[array_like] Array to check for masked values.

**Returns**

**result**

[bool] True if *x* is a MaskedArray with masked values, False otherwise.

## Examples

```
>>> import numpy.ma as ma
>>> x = ma.masked_equal([0, 1, 0, 2, 3], 0)
>>> x
masked_array(data=[--, 1, --, 2, 3],
             mask=[ True, False,  True, False, False],
       fill_value=0)
>>> ma.is_masked(x)
True
```

(continues on next page)

(continued from previous page)

```
>>> x = ma.masked_equal([0, 1, 0, 2, 3], 42)
>>> x
masked_array(data=[0, 1, 0, 2, 3],
             mask=False,
       fill_value=42)
>>> ma.is_masked(x)
False
```

Always returns False if *x* isn't a MaskedArray.

```
>>> x = [False, True, False]
>>> ma.is_masked(x)
False
>>> x = 'a string'
>>> ma.is_masked(x)
False
```

`numpy.ma.`**`is_mask`**`(`*m*`)`

Return True if m is a valid, standard mask.

This function does not check the contents of the input, only that the type is MaskType. In particular, this function returns False if the mask has a flexible dtype.

**Parameters**

**m**

[array_like] Array to test.

**Returns**

**result**

[bool] True if *m.dtype.type* is MaskType, False otherwise.

**See also:**

**`isMaskedArray`**

Test whether input is an instance of MaskedArray.

## Examples

```
>>> import numpy.ma as ma
>>> m = ma.masked_equal([0, 1, 0, 2, 3], 0)
>>> m
masked_array(data=[--, 1, --, 2, 3],
             mask=[ True, False,  True, False, False],
       fill_value=0)
>>> ma.is_mask(m)
False
>>> ma.is_mask(m.mask)
True
```

Input must be an ndarray (or have similar attributes) for it to be considered a valid mask.

```
>>> m = [False, True, False]
>>> ma.is_mask(m)
False
>>> m = np.array([False, True, False])
>>> m
array([False,  True, False])
>>> ma.is_mask(m)
True
```

Arrays with complex dtypes don't return True.

```
>>> dtype = np.dtype({'names':['monty', 'pithon'],
...                   'formats':[bool, bool]})
>>> dtype
dtype([('monty', '|b1'), ('pithon', '|b1')])
>>> m = np.array([(True, False), (False, True), (True, False)],
...              dtype=dtype)
>>> m
array([( True, False), (False,  True), ( True, False)],
      dtype=[('monty', '?'), ('pithon', '?')])
>>> ma.is_mask(m)
False
```

| | |
|---|---|
| *ma.MaskedArray.data* | Returns the underlying data, as a view of the masked array. |
| *ma.MaskedArray.mask* | Current mask. |
| *ma.MaskedArray.recordmask* | Get or set the mask of the array if it has no named fields. |

---

## Manipulating a MaskedArray

### Changing the shape

| | |
|---|---|
| *ma.ravel*(self[, order]) | Returns a 1D version of self, as a view. |
| *ma.reshape*(a, new_shape[, order]) | Returns an array containing the same data with a new shape. |
| *ma.resize*(x, new_shape) | Return a new masked array with the specified size and shape. |
| *ma.MaskedArray.flatten*([order]) | Return a copy of the array collapsed into one dimension. |
| *ma.MaskedArray.ravel*(self[, order]) | Returns a 1D version of self, as a view. |
| *ma.MaskedArray.reshape*(self, *s, **kwargs) | Give a new shape to the array without changing its data. |
| *ma.MaskedArray.resize*(self, newshape[, …]) | |

`numpy.ma.`**`ravel`**`(`*self*, *order='C'*`)` **`= <numpy.ma.core._frommethod object>`**

Returns a 1D version of self, as a view.

**Parameters**

**order**

[{'C', 'F', 'A', 'K'}, optional] The elements of *a* are read using this index order. 'C' means to index the elements in C-like order, with the last axis index changing fastest, back to the first axis index changing slowest. 'F' means to index the elements in Fortran-like index order, with the first index changing fastest, and the last index changing slowest. Note that the 'C' and 'F' options take no account of the memory layout of the underlying array, and only refer to the order of axis indexing. 'A' means to read the elements in Fortran-like index order if *m* is Fortran *contiguous* in memory, C-like order otherwise. 'K' means to read the elements in the order they occur in memory, except for reversing the data when strides are negative. By default, 'C' index order is used.

**Returns**

**MaskedArray**

Output view is of shape `(self.size,)` (or `(np.ma.product(self.shape),)`).

### Examples

```
>>> x = np.ma.array([[1,2,3],[4,5,6],[7,8,9]], mask=[0] + [1,0]*4)
>>> x
masked_array(
  data=[[1, --, 3],
        [--, 5, --],
        [7, --, 9]],
  mask=[[False,  True, False],
        [ True, False,  True],
        [False,  True, False]],
  fill_value=999999)
>>> x.ravel()
masked_array(data=[1, --, 3, --, 5, --, 7, --, 9],
             mask=[False,  True, False,  True, False,  True, False,  True,
                   False],
       fill_value=999999)
```

`numpy.ma.`**`reshape`**(*a*, *new_shape*, *order='C'*)

Returns an array containing the same data with a new shape.

Refer to *`MaskedArray.reshape`* for full documentation.

**See also:**

***`MaskedArray.reshape`***

equivalent function

`numpy.ma.`**`resize`**(*x*, *new_shape*)

Return a new masked array with the specified size and shape.

This is the masked equivalent of the *`numpy.resize`* function. The new array is filled with repeated copies of *x* (in the order that the data are stored in memory). If *x* is masked, the new array will be masked, and the new mask will be a repetition of the old one.

**See also:**

***`numpy.resize`***

Equivalent function in the top level NumPy module.

### Examples

```
>>> import numpy.ma as ma
>>> a = ma.array([[1, 2] ,[3, 4]])
>>> a[0, 1] = ma.masked
>>> a
masked_array(
  data=[[1, --],
        [3, 4]],
  mask=[[False,  True],
        [False, False]],
  fill_value=999999)
>>> np.resize(a, (3, 3))
masked_array(
  data=[[1, 2, 3],
        [4, 1, 2],
        [3, 4, 1]],
  mask=False,
  fill_value=999999)
>>> ma.resize(a, (3, 3))
masked_array(
  data=[[1, --, 3],
        [4, 1, --],
        [3, 4, 1]],
  mask=[[False,  True, False],
        [False, False,  True],
        [False, False, False]],
  fill_value=999999)
```

A MaskedArray is always returned, regardless of the input type.

```
>>> a = np.array([[1, 2] ,[3, 4]])
>>> ma.resize(a, (3, 3))
masked_array(
  data=[[1, 2, 3],
        [4, 1, 2],
        [3, 4, 1]],
  mask=False,
  fill_value=999999)
```

## Modifying axes

| | |
|---|---|
| `ma.swapaxes`(self, *args, …) | Return a view of the array with *axis1* and *axis2* interchanged. |
| `ma.transpose`(a[, axes]) | Permute the dimensions of an array. |
| `ma.MaskedArray.swapaxes`(axis1, axis2) | Return a view of the array with *axis1* and *axis2* interchanged. |
| `ma.MaskedArray.transpose`(*axes) | Returns a view of the array with axes transposed. |

`numpy.ma.`**`swapaxes`**(*self*, **args*, ***params*) *a.swapaxes(axis1, axis2)* **= <numpy.ma.core._frommethod object>**

Return a view of the array with *axis1* and *axis2* interchanged.

Refer to `numpy.swapaxes` for full documentation.

**See also:**

**`numpy.swapaxes`**

equivalent function

`numpy.ma.`**`transpose`**`(`*a*, *axes=None*`)`

Permute the dimensions of an array.

This function is exactly equivalent to `numpy.transpose`.

**See also:**

**`numpy.transpose`**

Equivalent function in top-level NumPy module.

### Examples

```
>>> import numpy.ma as ma
>>> x = ma.arange(4).reshape((2,2))
>>> x[1, 1] = ma.masked
>>> x
masked_array(
  data=[[0, 1],
        [2, --]],
  mask=[[False, False],
        [False,  True]],
  fill_value=999999)
```

```
>>> ma.transpose(x)
masked_array(
  data=[[0, 2],
        [1, --]],
  mask=[[False, False],
        [False,  True]],
  fill_value=999999)
```

## Changing the number of dimensions

| | |
|---|---|
| `ma.atleast_1d`(*args, **kwargs) | Convert inputs to arrays with at least one dimension. |
| `ma.atleast_2d`(*args, **kwargs) | View inputs as arrays with at least two dimensions. |
| `ma.atleast_3d`(*args, **kwargs) | View inputs as arrays with at least three dimensions. |
| `ma.expand_dims`(a, axis) | Expand the shape of an array. |
| `ma.squeeze`(a[, axis]) | Remove single-dimensional entries from the shape of an array. |
| `ma.MaskedArray.squeeze`([axis]) | Remove single-dimensional entries from the shape of *a*. |
| `ma.stack`(*args, **kwargs) | Join a sequence of arrays along a new axis. |
| `ma.column_stack`(*args, **kwargs) | Stack 1-D arrays as columns into a 2-D array. |
| `ma.concatenate`(arrays[, axis]) | Concatenate a sequence of arrays along the given axis. |
| `ma.dstack`(*args, **kwargs) | Stack arrays in sequence depth wise (along third axis). |
| `ma.hstack`(*args, **kwargs) | Stack arrays in sequence horizontally (column wise). |
| `ma.hsplit`(*args, **kwargs) | Split an array into multiple sub-arrays horizontally (column-wise). |
| `ma.mr_` | Translate slice objects to concatenation along the first axis. |

Continued on next page

Table 76 – continued from previous page

| | |
|---|---|
| `ma.row_stack`(*args, **kwargs) | Stack arrays in sequence vertically (row wise). |
| `ma.vstack`(*args, **kwargs) | Stack arrays in sequence vertically (row wise). |

numpy.ma.**atleast_1d**(**args*, ***kwargs*) **= <numpy.ma.extras._fromnxfunction_allargs object>**

Convert inputs to arrays with at least one dimension.

Scalar inputs are converted to 1-dimensional arrays, whilst higher-dimensional inputs are preserved.

**Parameters**

**arys1, arys2, …**

[array_like] One or more input arrays.

**Returns**

**ret**

[ndarray] An array, or list of arrays, each with `a.ndim >= 1`. Copies are made only if necessary.

### Notes

The function is applied to both the _data and the _mask, if any.

### Examples

```
>>> np.atleast_1d(1.0)
array([1.])
```

```
>>> x = np.arange(9.0).reshape(3,3)
>>> np.atleast_1d(x)
array([[0., 1., 2.],
       [3., 4., 5.],
       [6., 7., 8.]])
>>> np.atleast_1d(x) is x
True
```

```
>>> np.atleast_1d(1, [3, 4])
[array([1]), array([3, 4])]
```

numpy.ma.**atleast_2d**(**args*, ***kwargs*) **= <numpy.ma.extras._fromnxfunction_allargs object>**

View inputs as arrays with at least two dimensions.

**Parameters**

**arys1, arys2, …**

[array_like] One or more array-like sequences. Non-array inputs are converted to arrays. Arrays that already have two or more dimensions are preserved.

**Returns**

**res, res2, …**

[ndarray] An array, or list of arrays, each with `a.ndim >= 2`. Copies are avoided where possible, and views with two or more dimensions are returned.

## Notes

The function is applied to both the _data and the _mask, if any.

## Examples

```
>>> np.atleast_2d(3.0)
array([[3.]])
```

```
>>> x = np.arange(3.0)
>>> np.atleast_2d(x)
array([[0., 1., 2.]])
>>> np.atleast_2d(x).base is x
True
```

```
>>> np.atleast_2d(1, [1, 2], [[1, 2]])
[array([[1]]), array([[1, 2]]), array([[1, 2]])]
```

`numpy.ma.`**`atleast_3d`**`(`**args*, ***kwargs*`)` **`= <numpy.ma.extras._fromnxfunction_allargs object>`**

View inputs as arrays with at least three dimensions.

**Parameters**

**arys1, arys2, …**

[array_like] One or more array-like sequences. Non-array inputs are converted to arrays. Arrays that already have three or more dimensions are preserved.

**Returns**

**res1, res2, …**

[ndarray] An array, or list of arrays, each with `a.ndim >= 3`. Copies are avoided where possible, and views with three or more dimensions are returned. For example, a 1-D array of shape `(N,)` becomes a view of shape `(1, N, 1)`, and a 2-D array of shape `(M, N)` becomes a view of shape `(M, N, 1)`.

## Notes

The function is applied to both the _data and the _mask, if any.

## Examples

```
>>> np.atleast_3d(3.0)
array([[[3.]]])
```

```
>>> x = np.arange(3.0)
>>> np.atleast_3d(x).shape
(1, 3, 1)
```

```
>>> x = np.arange(12.0).reshape(4,3)
>>> np.atleast_3d(x).shape
(4, 3, 1)
>>> np.atleast_3d(x).base is x.base  # x is a reshape, so not base itself
True
```

```
>>> for arr in np.atleast_3d([1, 2], [[1, 2]], [[[1, 2]]]):
...     print(arr, arr.shape)
...
[[[1]
  [2]]] (1, 2, 1)
[[[1]
  [2]]] (1, 2, 1)
[[[1 2]]] (1, 1, 2)
```

`numpy.ma.`**`expand_dims`**(*a*, *axis*)

Expand the shape of an array.

Insert a new axis that will appear at the *axis* position in the expanded array shape.

**Parameters**

**a**

[array_like] Input array.

**axis**

[int or tuple of ints] Position in the expanded axes where the new axis (or axes) is placed.

Deprecated since version 1.13.0: Passing an axis where `axis > a.ndim` will be treated as `axis == a.ndim`, and passing `axis < -a.ndim - 1` will be treated as `axis == 0`. This behavior is deprecated.

Changed in version 1.18.0: A tuple of axes is now supported. Out of range axes as described above are now forbidden and raise an *AxisError*.

**Returns**

**result**

[ndarray] View of *a* with the number of dimensions increased.

**See also:**

***squeeze***

The inverse operation, removing singleton dimensions

***reshape***

Insert, remove, and combine dimensions, and resize existing ones

`doc.indexing`, *`atleast_1d`*, *`atleast_2d`*, *`atleast_3d`*

## Examples

```
>>> x = np.array([1, 2])
>>> x.shape
(2,)
```

The following is equivalent to `x[np.newaxis, :]` or `x[np.newaxis]`:

```
>>> y = np.expand_dims(x, axis=0)
>>> y
array([[1, 2]])
>>> y.shape
(1, 2)
```

The following is equivalent to `x[:, np.newaxis]`:

```
>>> y = np.expand_dims(x, axis=1)
>>> y
array([[1],
       [2]])
>>> y.shape
(2, 1)
```

`axis` may also be a tuple:

```
>>> y = np.expand_dims(x, axis=(0, 1))
>>> y
array([[[1, 2]]])
```

```
>>> y = np.expand_dims(x, axis=(2, 0))
>>> y
array([[[1],
         [2]]])
```

Note that some examples may use `None` instead of `np.newaxis`. These are the same objects:

```
>>> np.newaxis is None
True
```

`numpy.ma.`**`squeeze`**`(`*a*, *axis=None*`)`

Remove single-dimensional entries from the shape of an array.

**Parameters**

**a**

[array_like] Input data.

**axis**

[None or int or tuple of ints, optional] New in version 1.7.0.

Selects a subset of the single-dimensional entries in the shape. If an axis is selected with shape entry greater than one, an error is raised.

**Returns**

**squeezed**

[ndarray] The input array, but with all or a subset of the dimensions of length 1 removed. This is always *a* itself or a view into *a*. Note that if all axes are squeezed, the result is a 0d array and not a scalar.

**Raises**

**ValueError**

If *axis* is not None, and an axis being squeezed is not of length 1

**See also:**

***expand_dims***

The inverse operation, adding singleton dimensions

***reshape***

Insert, remove, and combine dimensions, and resize existing ones

### Examples

```
>>> x = np.array([[[0], [1], [2]]])
>>> x.shape
(1, 3, 1)
>>> np.squeeze(x).shape
(3,)
>>> np.squeeze(x, axis=0).shape
(3, 1)
>>> np.squeeze(x, axis=1).shape
Traceback (most recent call last):
...
ValueError: cannot select an axis to squeeze out which has size not equal to one
>>> np.squeeze(x, axis=2).shape
(1, 3)
>>> x = np.array([[1234]])
>>> x.shape
(1, 1)
>>> np.squeeze(x)
array(1234)  # 0d array
>>> np.squeeze(x).shape
()
>>> np.squeeze(x)[()]
1234
```

`numpy.ma.`**`stack`**`(`*`*args, **kwargs`*`)` **`= <numpy.ma.extras._fromnxfunction_seq object>`**

Join a sequence of arrays along a new axis.

The `axis` parameter specifies the index of the new axis in the dimensions of the result. For example, if `axis=0` it will be the first dimension and if `axis=-1` it will be the last dimension.

New in version 1.10.0.

**Parameters**

**arrays**

[sequence of array_like]

Each array must have the same shape.

**axis**

[int, optional] The axis in the result array along which the input arrays are stacked.

**out**

[ndarray, optional] If provided, the destination to place the result. The shape must be correct, matching that of what stack would have returned if no out argument were specified.

**Returns**

**stacked**

[ndarray] The stacked array has one more dimension than the input arrays.

**See also:**

**`concatenate`**

Join a sequence of arrays along an existing axis.

**`block`**

Assemble an nd-array from nested lists of blocks.

**`split`**

Split array into a list of multiple sub-arrays of equal size.

## Notes

The function is applied to both the _data and the _mask, if any.

## Examples

```
>>> arrays = [np.random.randn(3, 4) for _ in range(10)]
>>> np.stack(arrays, axis=0).shape
(10, 3, 4)
```

```
>>> np.stack(arrays, axis=1).shape
(3, 10, 4)
```

```
>>> np.stack(arrays, axis=2).shape
(3, 4, 10)
```

```
>>> a = np.array([1, 2, 3])
>>> b = np.array([2, 3, 4])
>>> np.stack((a, b))
array([[1, 2, 3],
       [2, 3, 4]])
```

```
>>> np.stack((a, b), axis=-1)
array([[1, 2],
       [2, 3],
       [3, 4]])
```

`numpy.ma.`**`column_stack`**`(`*`*args, **kwargs`*`)` **`= <numpy.ma.extras._fromnxfunction_seq object>`**

Stack 1-D arrays as columns into a 2-D array.

Take a sequence of 1-D arrays and stack them as columns to make a single 2-D array. 2-D arrays are stacked as-is, just like with `hstack`. 1-D arrays are turned into 2-D columns first.

**Parameters**

**tup**

[sequence of 1-D or 2-D arrays.] Arrays to stack. All of them must have the same first dimension.

**Returns**

**stacked**

[2-D array] The array formed by stacking the given arrays.

### Notes

The function is applied to both the _data and the _mask, if any.

### Examples

```
>>> a = np.array((1,2,3))
>>> b = np.array((2,3,4))
>>> np.column_stack((a,b))
array([[1, 2],
       [2, 3],
       [3, 4]])
```

`numpy.ma.`**`concatenate`**`(`*`arrays, axis=0`*`)`

Concatenate a sequence of arrays along the given axis.

**Parameters**

**arrays**

[sequence of array_like] The arrays must have the same shape, except in the dimension corresponding to *axis* (the first, by default).

**axis**

[int, optional] The axis along which the arrays will be joined. Default is 0.

**Returns**

**result**

[MaskedArray] The concatenated array with any masked entries preserved.

**See also:**

***numpy.concatenate***

Equivalent function in the top-level NumPy module.

## Examples

```
>>> import numpy.ma as ma
>>> a = ma.arange(3)
>>> a[1] = ma.masked
>>> b = ma.arange(2, 5)
>>> a
masked_array(data=[0, --, 2],
             mask=[False,  True, False],
       fill_value=999999)
>>> b
masked_array(data=[2, 3, 4],
             mask=False,
       fill_value=999999)
>>> ma.concatenate([a, b])
masked_array(data=[0, --, 2, 2, 3, 4],
             mask=[False,  True, False, False, False, False],
       fill_value=999999)
```

`numpy.ma.`**`dstack`**`(`**args*, ***kwargs*`)` **`= <numpy.ma.extras._fromnxfunction_seq object>`**

Stack arrays in sequence depth wise (along third axis).

This is equivalent to concatenation along the third axis after 2-D arrays of shape *(M,N)* have been reshaped to *(M,N,1)* and 1-D arrays of shape *(N,)* have been reshaped to *(1,N,1)*. Rebuilds arrays divided by *dsplit*.

This function makes most sense for arrays with up to 3 dimensions. For instance, for pixel-data with a height (first axis), width (second axis), and r/g/b channels (third axis). The functions `concatenate`, `stack` and *block* provide more general stacking and concatenation operations.

**Parameters**

**tup**

[sequence of arrays] The arrays must have the same shape along all but the third axis. 1-D or 2-D arrays must have the same shape.

**Returns**

**stacked**

[ndarray] The array formed by stacking the given arrays, will be at least 3-D.

**See also:**

***concatenate***

Join a sequence of arrays along an existing axis.

***stack***

Join a sequence of arrays along a new axis.

**block**

Assemble an nd-array from nested lists of blocks.

***vstack***

Stack arrays in sequence vertically (row wise).

***hstack***

Stack arrays in sequence horizontally (column wise).

***column_stack***

Stack 1-D arrays as columns into a 2-D array.

**dsplit**

Split array along third axis.

## Notes

The function is applied to both the _data and the _mask, if any.

## Examples

```
>>> a = np.array((1,2,3))
>>> b = np.array((2,3,4))
>>> np.dstack((a,b))
array([[[1, 2],
        [2, 3],
        [3, 4]]])
```

```
>>> a = np.array([[1],[2],[3]])
>>> b = np.array([[2],[3],[4]])
>>> np.dstack((a,b))
array([[[1, 2]],
       [[2, 3]],
       [[3, 4]]])
```

`numpy.ma.`**`hstack`**`(`**args*, ***kwargs*`)` **`= <numpy.ma.extras._fromnxfunction_seq object>`**

Stack arrays in sequence horizontally (column wise).

This is equivalent to concatenation along the second axis, except for 1-D arrays where it concatenates along the first axis. Rebuilds arrays divided by `hsplit`.

This function makes most sense for arrays with up to 3 dimensions. For instance, for pixel-data with a height (first axis), width (second axis), and r/g/b channels (third axis). The functions `concatenate`, `stack` and *block* provide more general stacking and concatenation operations.

**Parameters**

**tup**

[sequence of ndarrays] The arrays must have the same shape along all but the second axis, except 1-D arrays which can be any length.

**Returns**

**stacked**

[ndarray] The array formed by stacking the given arrays.

**See also:**

**`concatenate`**

Join a sequence of arrays along an existing axis.

**`stack`**

Join a sequence of arrays along a new axis.

**`block`**

Assemble an nd-array from nested lists of blocks.

**`vstack`**

Stack arrays in sequence vertically (row wise).

**`dstack`**

Stack arrays in sequence depth wise (along third axis).

**`column_stack`**

Stack 1-D arrays as columns into a 2-D array.

**`hsplit`**

Split an array into multiple sub-arrays horizontally (column-wise).

### Notes

The function is applied to both the _data and the _mask, if any.

### Examples

```
>>> a = np.array((1,2,3))
>>> b = np.array((2,3,4))
>>> np.hstack((a,b))
array([1, 2, 3, 2, 3, 4])
>>> a = np.array([[1],[2],[3]])
>>> b = np.array([[2],[3],[4]])
>>> np.hstack((a,b))
array([[1, 2],
       [2, 3],
       [3, 4]])
```

`numpy.ma.`**`hsplit`**`(`**args*, ***kwargs*`)` **`= <numpy.ma.extras._fromnxfunction_single object>`**

Split an array into multiple sub-arrays horizontally (column-wise).

Please refer to the *split* documentation. *`hsplit`* is equivalent to *split* with `axis=1`, the array is always split along the second axis regardless of the array dimension.

**See also:**

**`split`**

Split an array into multiple sub-arrays of equal size.

### Notes

The function is applied to both the _data and the _mask, if any.

### Examples

```
>>> x = np.arange(16.0).reshape(4, 4)
>>> x
array([[ 0.,   1.,   2.,   3.],
       [ 4.,   5.,   6.,   7.],
       [ 8.,   9.,  10.,  11.],
       [12.,  13.,  14.,  15.]])
>>> np.hsplit(x, 2)
[array([[  0.,   1.],
       [  4.,   5.],
       [  8.,   9.],
       [12.,  13.]]),
 array([[  2.,   3.],
       [  6.,   7.],
       [10.,  11.],
       [14.,  15.]])]
>>> np.hsplit(x, np.array([3, 6]))
[array([[ 0.,   1.,   2.],
       [ 4.,   5.,   6.],
```

(continues on next page)

(continued from previous page)

```
       [ 8.,   9.,  10.],
       [12.,  13.,  14.]]),
 array([[ 3.],
       [ 7.],
       [11.],
       [15.]]),
 array([], shape=(4, 0), dtype=float64)]
```

With a higher dimensional array the split is still along the second axis.

```
>>> x = np.arange(8.0).reshape(2, 2, 2)
>>> x
array([[[0.,  1.],
        [2.,  3.]],
       [[4.,  5.],
        [6.,  7.]]])
>>> np.hsplit(x, 2)
[array([[[0.,  1.]],
       [[4.,  5.]]]),
 array([[[2.,  3.]],
       [[6.,  7.]]])]
```

`numpy.ma.`**`mr_`** **`= <numpy.ma.extras.mr_class object>`**

Translate slice objects to concatenation along the first axis.

This is the masked array version of *lib.index_tricks.RClass*.

**See also:**

`lib.index_tricks.RClass`

## Examples

```
>>> np.ma.mr_[np.ma.array([1,2,3]), 0, 0, np.ma.array([4,5,6])]
masked_array(data=[1, 2, 3, ..., 4, 5, 6],
             mask=False,
       fill_value=999999)
```

`numpy.ma.`**`row_stack`**`(`**args*, ***kwargs*`)` **`= <numpy.ma.extras._fromnxfunction_seq object>`**

Stack arrays in sequence vertically (row wise).

This is equivalent to concatenation along the first axis after 1-D arrays of shape *(N,)* have been reshaped to *(1,N)*. Rebuilds arrays divided by *vsplit*.

This function makes most sense for arrays with up to 3 dimensions. For instance, for pixel-data with a height (first axis), width (second axis), and r/g/b channels (third axis). The functions `concatenate`, `stack` and *block* provide more general stacking and concatenation operations.

**Parameters**

**tup**

[sequence of ndarrays] The arrays must have the same shape along all but the first axis. 1-D arrays must have the same length.

**Returns**

**stacked**

[ndarray] The array formed by stacking the given arrays, will be at least 2-D.

**See also:**

***concatenate***

Join a sequence of arrays along an existing axis.

***stack***

Join a sequence of arrays along a new axis.

**block**

Assemble an nd-array from nested lists of blocks.

***hstack***

Stack arrays in sequence horizontally (column wise).

***dstack***

Stack arrays in sequence depth wise (along third axis).

***column_stack***

Stack 1-D arrays as columns into a 2-D array.

**vsplit**

Split an array into multiple sub-arrays vertically (row-wise).

### Notes

The function is applied to both the _data and the _mask, if any.

### Examples

```
>>> a = np.array([1, 2, 3])
>>> b = np.array([2, 3, 4])
>>> np.vstack((a,b))
array([[1, 2, 3],
       [2, 3, 4]])
```

```
>>> a = np.array([[1], [2], [3]])
>>> b = np.array([[2], [3], [4]])
>>> np.vstack((a,b))
array([[1],
       [2],
       [3],
       [2],
       [3],
       [4]])
```

`numpy.ma.`**`vstack`**`(`*`*args, **kwargs`*`)` **`= <numpy.ma.extras._fromnxfunction_seq object>`**

Stack arrays in sequence vertically (row wise).

This is equivalent to concatenation along the first axis after 1-D arrays of shape *(N,)* have been reshaped to *(1,N)*. Rebuilds arrays divided by *vsplit*.

This function makes most sense for arrays with up to 3 dimensions. For instance, for pixel-data with a height (first axis), width (second axis), and r/g/b channels (third axis). The functions `concatenate`, `stack` and *block* provide more general stacking and concatenation operations.

**Parameters**

**tup**

[sequence of ndarrays] The arrays must have the same shape along all but the first axis. 1-D arrays must have the same length.

**Returns**

**stacked**

[ndarray] The array formed by stacking the given arrays, will be at least 2-D.

**See also:**

**`concatenate`**

Join a sequence of arrays along an existing axis.

**`stack`**

Join a sequence of arrays along a new axis.

**`block`**

Assemble an nd-array from nested lists of blocks.

**`hstack`**

Stack arrays in sequence horizontally (column wise).

**`dstack`**

Stack arrays in sequence depth wise (along third axis).

**`column_stack`**

Stack 1-D arrays as columns into a 2-D array.

**`vsplit`**

Split an array into multiple sub-arrays vertically (row-wise).

### Notes

The function is applied to both the _data and the _mask, if any.

### Examples

```
>>> a = np.array([1, 2, 3])
>>> b = np.array([2, 3, 4])
>>> np.vstack((a,b))
array([[1, 2, 3],
       [2, 3, 4]])
```

```
>>> a = np.array([[1], [2], [3]])
>>> b = np.array([[2], [3], [4]])
>>> np.vstack((a,b))
array([[1],
       [2],
       [3],
       [2],
       [3],
       [4]])
```

## Joining arrays

| | |
|---|---|
| `ma.concatenate`(arrays[, axis]) | Concatenate a sequence of arrays along the given axis. |
| `ma.stack`(*args, **kwargs) | Join a sequence of arrays along a new axis. |
| `ma.vstack`(*args, **kwargs) | Stack arrays in sequence vertically (row wise). |
| `ma.hstack`(*args, **kwargs) | Stack arrays in sequence horizontally (column wise). |
| `ma.dstack`(*args, **kwargs) | Stack arrays in sequence depth wise (along third axis). |
| `ma.column_stack`(*args, **kwargs) | Stack 1-D arrays as columns into a 2-D array. |
| `ma.append`(a, b[, axis]) | Append values to the end of an array. |

`numpy.ma.`**`append`**(*a*, *b*, *axis=None*)

Append values to the end of an array.

New in version 1.9.0.

**Parameters**

**a**

[array_like] Values are appended to a copy of this array.

**b**

[array_like] These values are appended to a copy of *a*. It must be of the correct shape (the same shape as *a*, excluding *axis*). If *axis* is not specified, *b* can be any shape and will be flattened before use.

**axis**

[int, optional] The axis along which *v* are appended. If *axis* is not given, both *a* and *b* are flattened before use.

**Returns**

**append**

[MaskedArray] A copy of *a* with *b* appended to *axis*. Note that `append` does not occur in-place: a new array is allocated and filled. If *axis* is None, the result is a flattened array.

**See also:**

`numpy.append`

Equivalent function in the top-level NumPy module.

### Examples

```
>>> import numpy.ma as ma
>>> a = ma.masked_values([1, 2, 3], 2)
>>> b = ma.masked_values([[4, 5, 6], [7, 8, 9]], 7)
>>> ma.append(a, b)
masked_array(data=[1, --, 3, 4, 5, 6, --, 8, 9],
             mask=[False,  True, False, False, False, False,  True, False,
                   False],
       fill_value=999999)
```

---

## Operations on masks

### Creating a mask

| | |
|---|---|
| `ma.make_mask`(m[, copy, shrink, dtype]) | Create a boolean mask from an array. |
| `ma.make_mask_none`(newshape[, dtype]) | Return a boolean mask of the given shape, filled with False. |
| `ma.mask_or`(m1, m2[, copy, shrink]) | Combine two masks with the `logical_or` operator. |
| `ma.make_mask_descr`(ndtype) | Construct a dtype description list from a given dtype. |

`numpy.ma.`**`make_mask`**(*m*, *copy=False*, *shrink=True*, *dtype=<class 'numpy.bool_'>*)

Create a boolean mask from an array.

Return *m* as a boolean mask, creating a copy if necessary or requested. The function can accept any sequence that is convertible to integers, or `nomask`. Does not require that contents must be 0s and 1s, values of 0 are interpreted as False, everything else as True.

**Parameters**

**m**

[array_like] Potential mask.

**copy**

[bool, optional] Whether to return a copy of *m* (True) or *m* itself (False).

**shrink**

[bool, optional] Whether to shrink *m* to `nomask` if all its values are False.

**dtype**

[dtype, optional] Data-type of the output mask. By default, the output mask has a dtype of MaskType (bool). If the dtype is flexible, each field has a boolean dtype. This is ignored when *m* is `nomask`, in which case `nomask` is always returned.

**Returns**

**result**

[ndarray] A boolean mask derived from *m*.

### Examples

```
>>> import numpy.ma as ma
>>> m = [True, False, True, True]
>>> ma.make_mask(m)
array([ True, False,  True,  True])
>>> m = [1, 0, 1, 1]
>>> ma.make_mask(m)
array([ True, False,  True,  True])
>>> m = [1, 0, 2, -3]
>>> ma.make_mask(m)
array([ True, False,  True,  True])
```

Effect of the *shrink* parameter.

```
>>> m = np.zeros(4)
>>> m
array([0., 0., 0., 0.])
>>> ma.make_mask(m)
False
>>> ma.make_mask(m, shrink=False)
array([False, False, False, False])
```

Using a flexible *dtype*.

```
>>> m = [1, 0, 1, 1]
>>> n = [0, 1, 0, 0]
>>> arr = []
>>> for man, mouse in zip(m, n):
...     arr.append((man, mouse))
>>> arr
[(1, 0), (0, 1), (1, 0), (1, 0)]
>>> dtype = np.dtype({'names':['man', 'mouse'],
...                   'formats':[np.int64, np.int64]})
>>> arr = np.array(arr, dtype=dtype)
>>> arr
array([(1, 0), (0, 1), (1, 0), (1, 0)],
      dtype=[('man', '<i8'), ('mouse', '<i8')])
>>> ma.make_mask(arr, dtype=dtype)
array([(True, False), (False, True), (True, False), (True, False)],
      dtype=[('man', '|b1'), ('mouse', '|b1')])
```

`numpy.ma.`**`make_mask_none`**(*newshape*, *dtype=None*)

Return a boolean mask of the given shape, filled with False.

This function returns a boolean ndarray with all entries False, that can be used in common mask manipulations. If a complex dtype is specified, the type of each field is converted to a boolean type.

**Parameters**

**newshape**

[tuple] A tuple indicating the shape of the mask.

**dtype**

[{None, dtype}, optional] If None, use a MaskType instance. Otherwise, use a new datatype with the same fields as *dtype*, converted to boolean types.

**Returns**

**result**

[ndarray] An ndarray of appropriate shape and dtype, filled with False.

**See also:**

***make_mask***

Create a boolean mask from an array.

***make_mask_descr***

Construct a dtype description list from a given dtype.

## Examples

```
>>> import numpy.ma as ma
>>> ma.make_mask_none((3,))
array([False, False, False])
```

Defining a more complex dtype.

```
>>> dtype = np.dtype({'names':['foo', 'bar'],
...                   'formats':[np.float32, np.int64]})
>>> dtype
dtype([('foo', '<f4'), ('bar', '<i8')])
>>> ma.make_mask_none((3,), dtype=dtype)
array([(False, False), (False, False), (False, False)],
      dtype=[('foo', '|b1'), ('bar', '|b1')])
```

`numpy.ma.`**`mask_or`**(*m1*, *m2*, *copy=False*, *shrink=True*)

Combine two masks with the `logical_or` operator.

The result may be a view on *m1* or *m2* if the other is `nomask` (i.e. False).

**Parameters**

**m1, m2**

[array_like] Input masks.

**copy**

[bool, optional] If copy is False and one of the inputs is `nomask`, return a view of the other input mask. Defaults to False.

**shrink**

[bool, optional] Whether to shrink the output to `nomask` if all its values are False. Defaults to True.

**Returns**

**mask**

[output mask] The result masks values that are masked in either *m1* or *m2*.

**Raises**

**ValueError**

If *m1* and *m2* have different flexible dtypes.

### Examples

```
>>> m1 = np.ma.make_mask([0, 1, 1, 0])
>>> m2 = np.ma.make_mask([1, 0, 0, 0])
>>> np.ma.mask_or(m1, m2)
array([ True,  True,  True, False])
```

`numpy.ma.`**`make_mask_descr`**`(`*ndtype*`)`

Construct a dtype description list from a given dtype.

Returns a new dtype object, with the type of all fields in *ndtype* to a boolean type. Field names are not altered.

**Parameters**

**ndtype**

[dtype] The dtype to convert.

**Returns**

**result**

[dtype] A dtype that looks like *ndtype*, the type of all fields is boolean.

### Examples

```
>>> import numpy.ma as ma
>>> dtype = np.dtype({'names':['foo', 'bar'],
...                   'formats':[np.float32, np.int64]})
>>> dtype
dtype([('foo', '<f4'), ('bar', '<i8')])
>>> ma.make_mask_descr(dtype)
dtype([('foo', '|b1'), ('bar', '|b1')])
>>> ma.make_mask_descr(np.float32)
dtype('bool')
```

### Accessing a mask

| | |
|---|---|
| `ma.getmask(a)` | Return the mask of a masked array, or nomask. |
| `ma.getmaskarray(arr)` | Return the mask of a masked array, or full boolean array of False. |
| `ma.masked_array.mask` | Current mask. |

property

**property** masked_array.**mask**

Current mask.

### Finding masked data

| | |
|---|---|
| `ma.flatnotmasked_contiguous(a)` | Find contiguous unmasked data in a masked array along the given axis. |
| `ma.flatnotmasked_edges(a)` | Find the indices of the first and last unmasked values. |
| `ma.notmasked_contiguous(a[, axis])` | Find contiguous unmasked data in a masked array along the given axis. |
| `ma.notmasked_edges(a[, axis])` | Find the indices of the first and last unmasked values along an axis. |
| `ma.clump_masked(a)` | Returns a list of slices corresponding to the masked clumps of a 1-D array. |
| `ma.clump_unmasked(a)` | Return list of slices corresponding to the unmasked clumps of a 1-D array. |

numpy.ma.**flatnotmasked_contiguous**(*a*)

Find contiguous unmasked data in a masked array along the given axis.

**Parameters**

**a**

[narray] The input array.

**Returns**

**slice_list**

[list] A sorted sequence of *slice* objects (start index, end index).

**..versionchanged:: 1.15.0**

Now returns an empty list instead of None for a fully masked array

**See also:**

*flatnotmasked_edges*, *notmasked_contiguous*, *notmasked_edges*, *clump_masked*, *clump_unmasked*

## Notes

Only accepts 2-D arrays at most.

## Examples

```
>>> a = np.ma.arange(10)
>>> np.ma.flatnotmasked_contiguous(a)
[slice(0, 10, None)]
```

```
>>> mask = (a < 3) | (a > 8) | (a == 5)
>>> a[mask] = np.ma.masked
>>> np.array(a[~a.mask])
array([3, 4, 6, 7, 8])
```

```
>>> np.ma.flatnotmasked_contiguous(a)
[slice(3, 5, None), slice(6, 9, None)]
>>> a[:] = np.ma.masked
>>> np.ma.flatnotmasked_contiguous(a)
[]
```

`numpy.ma.`**`flatnotmasked_edges`**`(`*a*`)`

Find the indices of the first and last unmasked values.

Expects a 1-D *`MaskedArray`*, returns None if all values are masked.

**Parameters**

**a**

[array_like] Input 1-D *`MaskedArray`*

**Returns**

**edges**

[ndarray or None] The indices of first and last non-masked value in the array. Returns None if all values are masked.

**See also:**

*`flatnotmasked_contiguous`*, *`notmasked_contiguous`*, *`notmasked_edges`*, *`clump_masked`*, *`clump_unmasked`*

## Notes

Only accepts 1-D arrays.

## Examples

```
>>> a = np.ma.arange(10)
>>> np.ma.flatnotmasked_edges(a)
array([0, 9])
```

```
>>> mask = (a < 3) | (a > 8) | (a == 5)
>>> a[mask] = np.ma.masked
>>> np.array(a[~a.mask])
array([3, 4, 6, 7, 8])
```

```
>>> np.ma.flatnotmasked_edges(a)
array([3, 8])
```

```
>>> a[:] = np.ma.masked
>>> print(np.ma.flatnotmasked_edges(a))
None
```

`numpy.ma.`**`notmasked_contiguous`**(*a*, *axis=None*)

Find contiguous unmasked data in a masked array along the given axis.

**Parameters**

**a**

[array_like] The input array.

**axis**

[int, optional] Axis along which to perform the operation. If None (default), applies to a flattened version of the array, and this is the same as *`flatnotmasked_contiguous`*.

**Returns**

**endpoints**

[list] A list of slices (start and end indexes) of unmasked indexes in the array.

If the input is 2d and axis is specified, the result is a list of lists.

**See also:**

*`flatnotmasked_edges`*, *`flatnotmasked_contiguous`*, *`notmasked_edges`*, *`clump_masked`*, *`clump_unmasked`*

## Notes

Only accepts 2-D arrays at most.

### Examples

```
>>> a = np.arange(12).reshape((3, 4))
>>> mask = np.zeros_like(a)
>>> mask[1:, :-1] = 1; mask[0, 1] = 1; mask[-1, 0] = 0
>>> ma = np.ma.array(a, mask=mask)
>>> ma
masked_array(
  data=[[0, --, 2, 3],
        [--, --, --, 7],
        [8, --, --, 11]],
  mask=[[False,  True, False, False],
        [ True,  True,  True, False],
        [False,  True,  True, False]],
  fill_value=999999)
>>> np.array(ma[~ma.mask])
array([ 0,  2,  3,  7, 8, 11])
```

```
>>> np.ma.notmasked_contiguous(ma)
[slice(0, 1, None), slice(2, 4, None), slice(7, 9, None), slice(11, 12, None)]
```

```
>>> np.ma.notmasked_contiguous(ma, axis=0)
[[slice(0, 1, None), slice(2, 3, None)], [], [slice(0, 1, None)], [slice(0, 3,␣
↪None)]]
```

```
>>> np.ma.notmasked_contiguous(ma, axis=1)
[[slice(0, 1, None), slice(2, 4, None)], [slice(3, 4, None)], [slice(0, 1, None),␣
↪slice(3, 4, None)]]
```

`numpy.ma.`**`notmasked_edges`**(*a*, *axis=None*)

Find the indices of the first and last unmasked values along an axis.

If all values are masked, return None. Otherwise, return a list of two tuples, corresponding to the indices of the first and last unmasked values respectively.

**Parameters**

**a**

[array_like] The input array.

**axis**

[int, optional] Axis along which to perform the operation. If None (default), applies to a flattened version of the array.

**Returns**

**edges**

[ndarray or list] An array of start and end indexes if there are any masked data in the array. If there are no masked data in the array, *edges* is a list of the first and last index.

**See also:**

*flatnotmasked_contiguous*, *flatnotmasked_edges*, *notmasked_contiguous*, *clump_masked*, *clump_unmasked*

## Examples

```
>>> a = np.arange(9).reshape((3, 3))
>>> m = np.zeros_like(a)
>>> m[1:, 1:] = 1
```

```
>>> am = np.ma.array(a, mask=m)
>>> np.array(am[~am.mask])
array([0, 1, 2, 3, 6])
```

```
>>> np.ma.notmasked_edges(am)
array([0, 6])
```

`numpy.ma.`**`clump_masked`**`(`*a*`)`

Returns a list of slices corresponding to the masked clumps of a 1-D array. (A "clump" is defined as a contiguous region of the array).

**Parameters**

**a**

[ndarray] A one-dimensional masked array.

**Returns**

**slices**

[list of slice] The list of slices, one for each continuous region of masked elements in *a*.

**See also:**

*flatnotmasked_edges*, *flatnotmasked_contiguous*, *notmasked_edges*, *notmasked_contiguous*, *clump_unmasked*

## Notes

New in version 1.4.0.

## Examples

```
>>> a = np.ma.masked_array(np.arange(10))
>>> a[[0, 1, 2, 6, 8, 9]] = np.ma.masked
>>> np.ma.clump_masked(a)
[slice(0, 3, None), slice(6, 7, None), slice(8, 10, None)]
```

`numpy.ma.`**`clump_unmasked`**`(`*a*`)`

Return list of slices corresponding to the unmasked clumps of a 1-D array. (A "clump" is defined as a contiguous region of the array).

**Parameters**

**a**

[ndarray] A one-dimensional masked array.

**Returns**

**slices**

[list of slice] The list of slices, one for each continuous region of unmasked elements in *a*.

**See also:**

*flatnotmasked_edges*, *flatnotmasked_contiguous*, *notmasked_edges*, *notmasked_contiguous*, *clump_masked*

### Notes

New in version 1.4.0.

### Examples

```
>>> a = np.ma.masked_array(np.arange(10))
>>> a[[0, 1, 2, 6, 8, 9]] = np.ma.masked
>>> np.ma.clump_unmasked(a)
[slice(3, 6, None), slice(7, 8, None)]
```

## Modifying a mask

| | |
|---|---|
| *ma.mask_cols*(a[, axis]) | Mask columns of a 2D array that contain masked values. |
| *ma.mask_or*(m1, m2[, copy, shrink]) | Combine two masks with the `logical_or` operator. |
| *ma.mask_rowcols*(a[, axis]) | Mask rows and/or columns of a 2D array that contain masked values. |
| *ma.mask_rows*(a[, axis]) | Mask rows of a 2D array that contain masked values. |
| *ma.harden_mask*(self) | Force the mask to hard. |
| *ma.soften_mask*(self) | Force the mask to soft. |
| *ma.MaskedArray.harden_mask*(self) | Force the mask to hard. |
| *ma.MaskedArray.soften_mask*(self) | Force the mask to soft. |
| *ma.MaskedArray.shrink_mask*(self) | Reduce a mask to nomask when possible. |
| *ma.MaskedArray.unshare_mask*(self) | Copy the mask and set the sharedmask flag to False. |

`numpy.ma.`**`mask_cols`**`(`*a*, *axis=<no value>*`)`

Mask columns of a 2D array that contain masked values.

This function is a shortcut to `mask_rowcols` with *axis* equal to 1.

**See also:**

***mask_rowcols***

Mask rows and/or columns of a 2D array.

***masked_where***

Mask where a condition is met.

### Examples

```
>>> import numpy.ma as ma
>>> a = np.zeros((3, 3), dtype=int)
>>> a[1, 1] = 1
>>> a
array([[0, 0, 0],
       [0, 1, 0],
       [0, 0, 0]])
>>> a = ma.masked_equal(a, 1)
>>> a
masked_array(
  data=[[0, 0, 0],
        [0, --, 0],
        [0, 0, 0]],
  mask=[[False, False, False],
        [False,  True, False],
        [False, False, False]],
  fill_value=1)
>>> ma.mask_cols(a)
masked_array(
  data=[[0, --, 0],
        [0, --, 0],
        [0, --, 0]],
  mask=[[False,  True, False],
        [False,  True, False],
        [False,  True, False]],
  fill_value=1)
```

`numpy.ma.`**`mask_rowcols`**(*a*, *axis=None*)

Mask rows and/or columns of a 2D array that contain masked values.

Mask whole rows and/or columns of a 2D array that contain masked values. The masking behavior is selected using the *axis* parameter.

- If *axis* is None, rows *and* columns are masked.
- If *axis* is 0, only rows are masked.
- If *axis* is 1 or -1, only columns are masked.

**Parameters**

**a**

[array_like, MaskedArray] The array to mask. If not a MaskedArray instance (or if no array elements are masked). The result is a MaskedArray with *mask* set to `nomask` (False). Must be a 2D array.

**axis**

[int, optional] Axis along which to perform the operation. If None, applies to a flattened version of the array.

**Returns**

**a**

[MaskedArray] A modified version of the input array, masked depending on the value of the *axis* parameter.

**Raises**

**NotImplementedError**

If input array *a* is not 2D.

**See also:**

***mask_rows***

Mask rows of a 2D array that contain masked values.

***mask_cols***

Mask cols of a 2D array that contain masked values.

***masked_where***

Mask where a condition is met.

### Notes

The input array's mask is modified by this function.

### Examples

```
>>> import numpy.ma as ma
>>> a = np.zeros((3, 3), dtype=int)
>>> a[1, 1] = 1
>>> a
array([[0, 0, 0],
       [0, 1, 0],
       [0, 0, 0]])
>>> a = ma.masked_equal(a, 1)
>>> a
masked_array(
  data=[[0, 0, 0],
        [0, --, 0],
        [0, 0, 0]],
  mask=[[False, False, False],
        [False,  True, False],
        [False, False, False]],
  fill_value=1)
>>> ma.mask_rowcols(a)
masked_array(
  data=[[0, --, 0],
        [--, --, --],
        [0, --, 0]],
  mask=[[False,  True, False],
        [ True,  True,  True],
        [False,  True, False]],
  fill_value=1)
```

`numpy.ma.`**`mask_rows`**(*a*, *axis=<no value>*)

Mask rows of a 2D array that contain masked values.

This function is a shortcut to `mask_rowcols` with *axis* equal to 0.

**See also:**

***mask_rowcols***

Mask rows and/or columns of a 2D array.

***masked_where***

Mask where a condition is met.

### Examples

```
>>> import numpy.ma as ma
>>> a = np.zeros((3, 3), dtype=int)
>>> a[1, 1] = 1
>>> a
array([[0, 0, 0],
       [0, 1, 0],
       [0, 0, 0]])
>>> a = ma.masked_equal(a, 1)
>>> a
masked_array(
  data=[[0, 0, 0],
        [0, --, 0],
        [0, 0, 0]],
  mask=[[False, False, False],
        [False,  True, False],
        [False, False, False]],
  fill_value=1)
```

```
>>> ma.mask_rows(a)
masked_array(
  data=[[0, 0, 0],
        [--, --, --],
        [0, 0, 0]],
  mask=[[False, False, False],
        [ True,  True,  True],
        [False, False, False]],
  fill_value=1)
```

`numpy.ma.`**`harden_mask`**(*self*) **`= <numpy.ma.core._frommethod object>`**

Force the mask to hard.

Whether the mask of a masked array is hard or soft is determined by its *hardmask* property. `harden_mask` sets *hardmask* to True.

**See also:**

`hardmask`

`numpy.ma.`**`soften_mask`**(*self*) **`= <numpy.ma.core._frommethod object>`**

Force the mask to soft.

Whether the mask of a masked array is hard or soft is determined by its *hardmask* property. `soften_mask` sets *hardmask* to False.

**See also:**

`hardmask`

### Conversion operations

#### > to a masked array

| | |
|---|---|
| *ma.asarray*(a[, dtype, order]) | Convert the input to a masked array of the given data-type. |
| *ma.asanyarray*(a[, dtype]) | Convert the input to a masked array, conserving subclasses. |
| *ma.fix_invalid*(a[, mask, copy, fill_value]) | Return input with invalid data masked and replaced by a fill value. |
| *ma.masked_equal*(x, value[, copy]) | Mask an array where equal to a given value. |
| *ma.masked_greater*(x, value[, copy]) | Mask an array where greater than a given value. |
| *ma.masked_greater_equal*(x, value[, copy]) | Mask an array where greater than or equal to a given value. |
| *ma.masked_inside*(x, v1, v2[, copy]) | Mask an array inside a given interval. |
| *ma.masked_invalid*(a[, copy]) | Mask an array where invalid values occur (NaNs or infs). |
| *ma.masked_less*(x, value[, copy]) | Mask an array where less than a given value. |
| *ma.masked_less_equal*(x, value[, copy]) | Mask an array where less than or equal to a given value. |
| *ma.masked_not_equal*(x, value[, copy]) | Mask an array where *not* equal to a given value. |
| *ma.masked_object*(x, value[, copy, shrink]) | Mask the array *x* where the data are exactly equal to value. |
| *ma.masked_outside*(x, v1, v2[, copy]) | Mask an array outside a given interval. |
| *ma.masked_values*(x, value[, rtol, atol, …]) | Mask using floating point equality. |
| *ma.masked_where*(condition, a[, copy]) | Mask an array where a condition is met. |

#### > to a ndarray

| | |
|---|---|
| *ma.compress_cols*(a) | Suppress whole columns of a 2-D array that contain masked values. |
| *ma.compress_rowcols*(x[, axis]) | Suppress the rows and/or columns of a 2-D array that contain masked values. |
| *ma.compress_rows*(a) | Suppress whole rows of a 2-D array that contain masked values. |
| *ma.compressed*(x) | Return all the non-masked data as a 1-D array. |
| *ma.filled*(a[, fill_value]) | Return input as an array with masked data replaced by a fill value. |
| *ma.MaskedArray.compressed*(self) | Return all the non-masked data as a 1-D array. |
| *ma.MaskedArray.filled*(self[, fill_value]) | Return a copy of self, with masked values filled with a given value. |

`numpy.ma.`**`compress_cols`**(*a*)

Suppress whole columns of a 2-D array that contain masked values.

This is equivalent to `np.ma.compress_rowcols(a, 1)`, see `extras.compress_rowcols` for details.

**See also:**

`extras.compress_rowcols`

`numpy.ma.`**`compress_rowcols`**(*x*, *axis=None*)

Suppress the rows and/or columns of a 2-D array that contain masked values.

The suppression behavior is selected with the *axis* parameter.

- If axis is None, both rows and columns are suppressed.
- If axis is 0, only rows are suppressed.
- If axis is 1 or -1, only columns are suppressed.

**Parameters**

**x**

[array_like, MaskedArray] The array to operate on. If not a MaskedArray instance (or if no array elements are masked), *x* is interpreted as a MaskedArray with *mask* set to `nomask`. Must be a 2D array.

**axis**

[int, optional] Axis along which to perform the operation. Default is None.

**Returns**

**compressed_array**

[ndarray] The compressed array.

## Examples

```
>>> x = np.ma.array(np.arange(9).reshape(3, 3), mask=[[1, 0, 0],
...                                                   [1, 0, 0],
...                                                   [0, 0, 0]])
>>> x
masked_array(
  data=[[--, 1, 2],
        [--, 4, 5],
        [6, 7, 8]],
  mask=[[ True, False, False],
        [ True, False, False],
        [False, False, False]],
  fill_value=999999)
```

```
>>> np.ma.compress_rowcols(x)
array([[7, 8]])
>>> np.ma.compress_rowcols(x, 0)
array([[6, 7, 8]])
>>> np.ma.compress_rowcols(x, 1)
array([[1, 2],
       [4, 5],
       [7, 8]])
```

`numpy.ma.`**`compress_rows`**`(`*a*`)`

Suppress whole rows of a 2-D array that contain masked values.

This is equivalent to `np.ma.compress_rowcols(a, 0)`, see `extras.compress_rowcols` for details.

**See also:**

`extras.compress_rowcols`

`numpy.ma.`**`compressed`**`(`*x*`)`

Return all the non-masked data as a 1-D array.

This function is equivalent to calling the "compressed" method of a `MaskedArray`, see `MaskedArray.compressed` for details.

**See also:**

***`MaskedArray.compressed`***

Equivalent method.

`numpy.ma.`**`filled`**`(`*a*, *fill_value=None*`)`

Return input as an array with masked data replaced by a fill value.

If *a* is not a `MaskedArray`, *a* itself is returned. If *a* is a `MaskedArray` and *fill_value* is None, *fill_value* is set to `a.fill_value`.

**Parameters**

**a**

[MaskedArray or array_like] An input object.

**fill_value**

[array_like, optional.] Can be scalar or non-scalar. If non-scalar, the resulting filled array should be broadcastable over input array. Default is None.

**Returns**

**a**

[ndarray] The filled array.

**See also:**

`compressed`

## Examples

```
>>> x = np.ma.array(np.arange(9).reshape(3, 3), mask=[[1, 0, 0],
...                                                   [1, 0, 0],
...                                                   [0, 0, 0]])
>>> x.filled()
array([[999999,      1,      2],
       [999999,      4,      5],
       [     6,      7,      8]])
>>> x.filled(fill_value=333)
array([[333,   1,   2],
       [333,   4,   5],
       [  6,   7,   8]])
>>> x.filled(fill_value=np.arange(3))
array([[0, 1, 2],
       [0, 4, 5],
       [6, 7, 8]])
```

### > to another object

| | |
|---|---|
| `ma.MaskedArray.tofile`(self, fid[, sep, format]) | Save a masked array to a file in binary format. |
| `ma.MaskedArray.tolist`(self[, fill_value]) | Return the data portion of the masked array as a hierarchical Python list. |
| `ma.MaskedArray.torecords`(self) | Transforms a masked array into a flexible-type array. |
| `ma.MaskedArray.tobytes`(self[, fill_value, order]) | Return the array data as a string containing the raw bytes in the array. |

### Filling a masked array

| | |
|---|---|
| `ma.common_fill_value`(a, b) | Return the common filling value of two masked arrays, if any. |
| `ma.default_fill_value`(obj) | Return the default fill value for the argument object. |
| `ma.maximum_fill_value`(obj) | Return the minimum value that can be represented by the dtype of an object. |
| `ma.maximum_fill_value`(obj) | Return the minimum value that can be represented by the dtype of an object. |
| `ma.set_fill_value`(a, fill_value) | Set the filling value of a, if a is a masked array. |
| `ma.MaskedArray.get_fill_value`(self) | The filling value of the masked array is a scalar. |
| `ma.MaskedArray.set_fill_value`(self[, value]) | |

`numpy.ma.`**`common_fill_value`**`(`*a*, *b*`)`

Return the common filling value of two masked arrays, if any.

If `a.fill_value == b.fill_value`, return the fill value, otherwise return None.

**Parameters**

**a, b**

[MaskedArray] The masked arrays for which to compare fill values.

**Returns**

**fill_value**

[scalar or None] The common fill value, or None.

### Examples

```
>>> x = np.ma.array([0, 1.], fill_value=3)
>>> y = np.ma.array([0, 1.], fill_value=3)
>>> np.ma.common_fill_value(x, y)
3.0
```

`numpy.ma.`**`default_fill_value`**`(`*obj*`)`

Return the default fill value for the argument object.

The default filling value depends on the datatype of the input array or the type of the input scalar:

| datatype | default |
|---|---|
| bool | True |
| int | 999999 |
| float | 1.e20 |
| complex | 1.e20+0j |
| object | '?' |
| string | 'N/A' |

For structured types, a structured scalar is returned, with each field the default fill value for its type.

For subarray types, the fill value is an array of the same size containing the default scalar fill value.

**Parameters**

**obj**

[ndarray, dtype or scalar] The array data-type or scalar for which the default fill value is returned.

**Returns**

**fill_value**

[scalar] The default fill value.

## Examples

```
>>> np.ma.default_fill_value(1)
999999
>>> np.ma.default_fill_value(np.array([1.1, 2., np.pi]))
1e+20
>>> np.ma.default_fill_value(np.dtype(complex))
(1e+20+0j)
```

`numpy.ma.`**`maximum_fill_value`**`(`*obj*`)`

Return the minimum value that can be represented by the dtype of an object.

This function is useful for calculating a fill value suitable for taking the maximum of an array with a given dtype.

**Parameters**

**obj**

[ndarray, dtype or scalar] An object that can be queried for it's numeric type.

**Returns**

**val**

[scalar] The minimum representable value.

**Raises**

**TypeError**

If *obj* isn't a suitable numeric type.

**See also:**

**`minimum_fill_value`**
The inverse function.

*`set_fill_value`*
Set the filling value of a masked array.

*`MaskedArray.fill_value`*
Return current fill value.

### Examples

```
>>> import numpy.ma as ma
>>> a = np.int8()
>>> ma.maximum_fill_value(a)
-128
>>> a = np.int32()
>>> ma.maximum_fill_value(a)
-2147483648
```

An array of numeric data can also be passed.

```
>>> a = np.array([1, 2, 3], dtype=np.int8)
>>> ma.maximum_fill_value(a)
-128
>>> a = np.array([1, 2, 3], dtype=np.float32)
>>> ma.maximum_fill_value(a)
-inf
```

`numpy.ma.`**`set_fill_value`**`(`*a*, *fill_value*`)`
Set the filling value of a, if a is a masked array.

This function changes the fill value of the masked array *a* in place. If *a* is not a masked array, the function returns silently, without doing anything.

**Parameters**

**a**

[array_like] Input array.

**fill_value**

[dtype] Filling value. A consistency test is performed to make sure the value is compatible with the dtype of *a*.

**Returns**

**None**

Nothing returned by this function.

**See also:**

***maximum_fill_value***

Return the default fill value for a dtype.

***MaskedArray.fill_value***

Return current fill value.

***MaskedArray.set_fill_value***

Equivalent method.

### Examples

```
>>> import numpy.ma as ma
>>> a = np.arange(5)
>>> a
array([0, 1, 2, 3, 4])
>>> a = ma.masked_where(a < 3, a)
>>> a
masked_array(data=[--, --, --, 3, 4],
             mask=[ True,  True,  True, False, False],
       fill_value=999999)
>>> ma.set_fill_value(a, -999)
>>> a
masked_array(data=[--, --, --, 3, 4],
             mask=[ True,  True,  True, False, False],
       fill_value=-999)
```

Nothing happens if *a* is not a masked array.

```
>>> a = list(range(5))
>>> a
[0, 1, 2, 3, 4]
>>> ma.set_fill_value(a, 100)
>>> a
[0, 1, 2, 3, 4]
>>> a = np.arange(5)
>>> a
array([0, 1, 2, 3, 4])
>>> ma.set_fill_value(a, 100)
>>> a
array([0, 1, 2, 3, 4])
```

| | |
|---|---|
| *ma.MaskedArray.fill_value* | The filling value of the masked array is a scalar. |

## Masked arrays arithmetics

### Arithmetics

| | |
|---|---|
| `ma.anom`(self[, axis, dtype]) | Compute the anomalies (deviations from the arithmetic mean) along the given axis. |
| `ma.anomalies`(self[, axis, dtype]) | Compute the anomalies (deviations from the arithmetic mean) along the given axis. |
| `ma.average`(a[, axis, weights, returned]) | Return the weighted average of array over the given axis. |
| `ma.conjugate`(x, /[, out, where, casting, …]) | Return the complex conjugate, element-wise. |
| `ma.corrcoef`(x[, y, rowvar, bias, …]) | Return Pearson product-moment correlation coefficients. |
| `ma.cov`(x[, y, rowvar, bias, allow_masked, ddof]) | Estimate the covariance matrix. |
| `ma.cumsum`(self[, axis, dtype, out]) | Return the cumulative sum of the array elements over the given axis. |
| `ma.cumprod`(self[, axis, dtype, out]) | Return the cumulative product of the array elements over the given axis. |
| `ma.mean`(self[, axis, dtype, out, keepdims]) | Returns the average of the array elements along given axis. |
| `ma.median`(a[, axis, out, overwrite_input, …]) | Compute the median along the specified axis. |
| `ma.power`(a, b[, third]) | Returns element-wise base array raised to power from second array. |
| `ma.prod`(self[, axis, dtype, out, keepdims]) | Return the product of the array elements over the given axis. |
| `ma.std`(self[, axis, dtype, out, ddof, keepdims]) | Returns the standard deviation of the array elements along given axis. |
| `ma.sum`(self[, axis, dtype, out, keepdims]) | Return the sum of the array elements over the given axis. |
| `ma.var`(self[, axis, dtype, out, ddof, keepdims]) | Compute the variance along the specified axis. |
| `ma.MaskedArray.anom`(self[, axis, dtype]) | Compute the anomalies (deviations from the arithmetic mean) along the given axis. |
| `ma.MaskedArray.cumprod`(self[, axis, dtype, out]) | Return the cumulative product of the array elements over the given axis. |
| `ma.MaskedArray.cumsum`(self[, axis, dtype, out]) | Return the cumulative sum of the array elements over the given axis. |
| `ma.MaskedArray.mean`(self[, axis, dtype, …]) | Returns the average of the array elements along given axis. |
| `ma.MaskedArray.prod`(self[, axis, dtype, …]) | Return the product of the array elements over the given axis. |
| `ma.MaskedArray.std`(self[, axis, dtype, out, …]) | Returns the standard deviation of the array elements along given axis. |
| `ma.MaskedArray.sum`(self[, axis, dtype, out, …]) | Return the sum of the array elements over the given axis. |
| `ma.MaskedArray.var`(self[, axis, dtype, out, …]) | Compute the variance along the specified axis. |

`numpy.ma.`**`anom`**(*self*, *axis=None*, *dtype=None*) **`= <numpy.ma.core._frommethod object>`**

Compute the anomalies (deviations from the arithmetic mean) along the given axis.

Returns an array of anomalies, with the same shape as the input and where the arithmetic mean is computed along the given axis.

**Parameters**

**axis**

[int, optional] Axis over which the anomalies are taken. The default is to use the mean of the flattened array as reference.

**dtype**

[dtype, optional]

**Type to use in computing the variance. For arrays of integer type**

the default is float32; for arrays of float types it is the same as the array type.

**See also:**

*mean*

Compute the mean of the array.

## Examples

```
>>> a = np.ma.array([1,2,3])
>>> a.anom()
masked_array(data=[-1.,  0.,  1.],
             mask=False,
       fill_value=1e+20)
```

`numpy.ma.`**`anomalies`**`(`*self*, *axis=None*, *dtype=None*`)` **`= <numpy.ma.core._frommethod object>`**

Compute the anomalies (deviations from the arithmetic mean) along the given axis.

Returns an array of anomalies, with the same shape as the input and where the arithmetic mean is computed along the given axis.

**Parameters**

**axis**

[int, optional] Axis over which the anomalies are taken. The default is to use the mean of the flattened array as reference.

**dtype**

[dtype, optional]

**Type to use in computing the variance. For arrays of integer type**

the default is float32; for arrays of float types it is the same as the array type.

**See also:**

*mean*

Compute the mean of the array.

## Examples

```
>>> a = np.ma.array([1,2,3])
>>> a.anom()
masked_array(data=[-1.,  0.,  1.],
             mask=False,
       fill_value=1e+20)
```

`numpy.ma.`**`average`**`(`*a*, *axis=None*, *weights=None*, *returned=False*`)`

Return the weighted average of array over the given axis.

**Parameters**

**a**

[array_like] Data to be averaged. Masked entries are not taken into account in the computation.

**axis**

[int, optional] Axis along which to average *a*. If None, averaging is done over the flattened array.

**weights**

[array_like, optional] The importance that each element has in the computation of the average. The weights array can either be 1-D (in which case its length must be the size of *a* along the given axis) or of the same shape as *a*. If `weights=None`, then all data in *a* are assumed to have a weight equal to one. The 1-D calculation is:

```
avg = sum(a * weights) / sum(weights)
```

The only constraint on *weights* is that *sum(weights)* must not be 0.

**returned**

[bool, optional] Flag indicating whether a tuple `(result, sum of weights)` should be returned as output (True), or just the result (False). Default is False.

**Returns**

**average, [sum_of_weights]**

[(tuple of) scalar or MaskedArray] The average along the specified axis. When returned is *True*, return a tuple with the average as the first element and the sum of the weights as the second element. The return type is *np.float64* if *a* is of integer type and floats smaller than *float64*, or the input data-type, otherwise. If returned, *sum_of_weights* is always *float64*.

## Examples

```
>>> a = np.ma.array([1., 2., 3., 4.], mask=[False, False, True, True])
>>> np.ma.average(a, weights=[3, 1, 0, 0])
1.25
```

```
>>> x = np.ma.arange(6.).reshape(3, 2)
>>> x
masked_array(
  data=[[0., 1.],
```

(continues on next page)

(continued from previous page)

```
        [2., 3.],
        [4., 5.]],
  mask=False,
  fill_value=1e+20)
>>> avg, sumweights = np.ma.average(x, axis=0, weights=[1, 2, 3],
...                                 returned=True)
>>> avg
masked_array(data=[2.6666666666666665, 3.6666666666666665],
             mask=[False, False],
       fill_value=1e+20)
```

`numpy.ma.`**`conjugate`**`(`*x, /, out=None, *, where=True, casting='same_kind', order='K', dtype=None, subok=True*[, *signature, extobj*]`)` **`= <numpy.ma.core._MaskedUnaryOperation object>`**

Return the complex conjugate, element-wise.

The complex conjugate of a complex number is obtained by changing the sign of its imaginary part.

**Parameters**

**x**

[array_like] Input value.

**out**

[ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

**where**

[array_like, optional] This condition is broadcast over the input. At locations where the condition is True, the *out* array will be set to the ufunc result. Elsewhere, the *out* array will retain its original value. Note that if an uninitialized *out* array is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.

****kwargs**

For other keyword-only arguments, see the *ufunc docs*.

**Returns**

**y**

[ndarray] The complex conjugate of *x*, with same dtype as *y*. This is a scalar if *x* is a scalar.

## Notes

*conj* is an alias for `conjugate`:

```
>>> np.conj is np.conjugate
True
```

## Examples

```
>>> np.conjugate(1+2j)
(1-2j)
```

```
>>> x = np.eye(2) + 1j * np.eye(2)
>>> np.conjugate(x)
array([[ 1.-1.j,  0.-0.j],
       [ 0.-0.j,  1.-1.j]])
```

`numpy.ma.`**`corrcoef`**(*x*, *y=None*, *rowvar=True*, *bias=<no value>*, *allow_masked=True*, *ddof=<no value>*)

Return Pearson product-moment correlation coefficients.

Except for the handling of missing data this function does the same as `numpy.corrcoef`. For more details and examples, see `numpy.corrcoef`.

**Parameters**

**x**

[array_like] A 1-D or 2-D array containing multiple variables and observations. Each row of *x* represents a variable, and each column a single observation of all those variables. Also see *rowvar* below.

**y**

[array_like, optional] An additional set of variables and observations. *y* has the same shape as *x*.

**rowvar**

[bool, optional] If *rowvar* is True (default), then each row represents a variable, with observations in the columns. Otherwise, the relationship is transposed: each column represents a variable, while the rows contain observations.

**bias**

[_NoValue, optional] Has no effect, do not use.

Deprecated since version 1.10.0.

**allow_masked**

[bool, optional] If True, masked values are propagated pair-wise: if a value is masked in *x*, the corresponding value is masked in *y*. If False, raises an exception. Because *bias* is deprecated, this argument needs to be treated as keyword only to avoid a warning.

**ddof**

[_NoValue, optional] Has no effect, do not use.

Deprecated since version 1.10.0.

**See also:**

***`numpy.corrcoef`***

Equivalent function in top-level NumPy module.

***`cov`***

Estimate the covariance matrix.

### Notes

This function accepts but discards arguments *bias* and *ddof*. This is for backwards compatibility with previous versions of this function. These arguments had no effect on the return values of the function and can be safely ignored in this and previous versions of numpy.

`numpy.ma.`**`cov`**(*x*, *y=None*, *rowvar=True*, *bias=False*, *allow_masked=True*, *ddof=None*)

Estimate the covariance matrix.

Except for the handling of missing data this function does the same as *`numpy.cov`*. For more details and examples, see *`numpy.cov`*.

By default, masked values are recognized as such. If *x* and *y* have the same shape, a common mask is allocated: if `x[i,j]` is masked, then `y[i,j]` will also be masked. Setting *allow_masked* to False will raise an exception if values are missing in either of the input arrays.

**Parameters**

**x**

[array_like] A 1-D or 2-D array containing multiple variables and observations. Each row of *x* represents a variable, and each column a single observation of all those variables. Also see *rowvar* below.

**y**

[array_like, optional] An additional set of variables and observations. *y* has the same form as *x*.

**rowvar**

[bool, optional] If *rowvar* is True (default), then each row represents a variable, with observations in the columns. Otherwise, the relationship is transposed: each column represents a variable, while the rows contain observations.

**bias**

[bool, optional] Default normalization (False) is by `(N-1)`, where `N` is the number of observations given (unbiased estimate). If *bias* is True, then normalization is by `N`. This keyword can be overridden by the keyword `ddof` in numpy versions >= 1.5.

**allow_masked**

[bool, optional] If True, masked values are propagated pair-wise: if a value is masked in *x*, the corresponding value is masked in *y*. If False, raises a *ValueError* exception when some values are missing.

**ddof**

[{None, int}, optional] If not `None` normalization is by `(N - ddof)`, where `N` is the number of observations; this overrides the value implied by `bias`. The default value is `None`.

New in version 1.5.

**Raises**

**ValueError**

Raised if some values are missing and *allow_masked* is False.

**See also:**

`numpy.cov`

`numpy.ma.`**`cumsum`**`(`*self*, *axis=None*, *dtype=None*, *out=None*`)` **`= <numpy.ma.core._frommethod object>`**

Return the cumulative sum of the array elements over the given axis.

Masked values are set to 0 internally during the computation. However, their position is saved, and the result will be masked at the same locations.

Refer to `numpy.cumsum` for full documentation.

**See also:**

***`numpy.ndarray.cumsum`***

corresponding function for ndarrays

***`numpy.cumsum`***

equivalent function

### Notes

The mask is lost if *out* is not a valid `MaskedArray` !

Arithmetic is modular when using integer types, and no error is raised on overflow.

### Examples

```
>>> marr = np.ma.array(np.arange(10), mask=[0,0,0,1,1,1,0,0,0,0])
>>> marr.cumsum()
masked_array(data=[0, 1, 3, --, --, --, 9, 16, 24, 33],
             mask=[False, False, False,  True,  True,  True, False, False,
                   False, False],
       fill_value=999999)
```

`numpy.ma.`**`cumprod`**`(`*self*, *axis=None*, *dtype=None*, *out=None*`)` **`= <numpy.ma.core._frommethod object>`**

Return the cumulative product of the array elements over the given axis.

Masked values are set to 1 internally during the computation. However, their position is saved, and the result will be masked at the same locations.

Refer to `numpy.cumprod` for full documentation.

**See also:**

***`numpy.ndarray.cumprod`***

corresponding function for ndarrays

***numpy.cumprod***

equivalent function

### Notes

The mask is lost if *out* is not a valid MaskedArray !

Arithmetic is modular when using integer types, and no error is raised on overflow.

`numpy.ma.`**`mean`**(*self*, *axis=None*, *dtype=None*, *out=None*, *keepdims=<no value>*) **`= <numpy.ma.core._frommethod object>`**

Returns the average of the array elements along given axis.

Masked entries are ignored, and result elements which are not finite will be masked.

Refer to *`numpy.mean`* for full documentation.

**See also:**

***numpy.ndarray.mean***

corresponding function for ndarrays

***numpy.mean***

Equivalent function

***numpy.ma.average***

Weighted average.

### Examples

```
>>> a = np.ma.array([1,2,3], mask=[False, False, True])
>>> a
masked_array(data=[1, 2, --],
             mask=[False, False,  True],
       fill_value=999999)
>>> a.mean()
1.5
```

`numpy.ma.`**`median`**(*a*, *axis=None*, *out=None*, *overwrite_input=False*, *keepdims=False*)

Compute the median along the specified axis.

Returns the median of the array elements.

**Parameters**

**a**

[array_like] Input array or object that can be converted to an array.

**axis**

[int, optional] Axis along which the medians are computed. The default (None) is to compute the median along a flattened version of the array.

**out**

[ndarray, optional] Alternative output array in which to place the result. It must have the same shape and buffer length as the expected output but the type will be cast if necessary.

**overwrite_input**

[bool, optional] If True, then allow use of memory of input array (a) for calculations. The input array will be modified by the call to median. This will save memory when you do not need to preserve the contents of the input array. Treat the input as undefined, but it will probably be fully or partially sorted. Default is False. Note that, if *overwrite_input* is True, and the input is not already an *ndarray*, an error will be raised.

**keepdims**

[bool, optional] If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the input array.

New in version 1.10.0.

**Returns**

**median**

[ndarray] A new array holding the result is returned unless out is specified, in which case a reference to out is returned. Return data-type is *float64* for integers and floats smaller than *float64*, or the input data-type, otherwise.

**See also:**

`mean`

## Notes

Given a vector `V` with `N` non masked values, the median of `V` is the middle value of a sorted copy of `V` (`Vs`) - i.e. `Vs[(N-1)/2]`, when `N` is odd, or `{Vs[N/2 - 1] + Vs[N/2]}/2` when `N` is even.

## Examples

```
>>> x = np.ma.array(np.arange(8), mask=[0]*4 + [1]*4)
>>> np.ma.median(x)
1.5
```

```
>>> x = np.ma.array(np.arange(10).reshape(2, 5), mask=[0]*6 + [1]*4)
>>> np.ma.median(x)
2.5
>>> np.ma.median(x, axis=-1, overwrite_input=True)
masked_array(data=[2.0, 5.0],
             mask=[False, False],
       fill_value=1e+20)
```

`numpy.ma.`**`power`**(*a*, *b*, *third=None*)

Returns element-wise base array raised to power from second array.

This is the masked array version of `numpy.power`. For details see `numpy.power`.

**See also:**

`numpy.power`

### Notes

The *out* argument to `numpy.power` is not supported, *third* has to be None.

`numpy.ma.`**`prod`**(*self*, *axis=None*, *dtype=None*, *out=None*, *keepdims=<no value>*) **`= <numpy.ma.core._frommethod object>`**

Return the product of the array elements over the given axis.

Masked elements are set to 1 internally for computation.

Refer to `numpy.prod` for full documentation.

**See also:**

***`numpy.ndarray.prod`***

corresponding function for ndarrays

***`numpy.prod`***

equivalent function

### Notes

Arithmetic is modular when using integer types, and no error is raised on overflow.

`numpy.ma.`**`std`**(*self*, *axis=None*, *dtype=None*, *out=None*, *ddof=0*, *keepdims=<no value>*) **`= <numpy.ma.core._frommethod object>`**

Returns the standard deviation of the array elements along given axis.

Masked entries are ignored.

Refer to `numpy.std` for full documentation.

**See also:**

***`numpy.ndarray.std`***

corresponding function for ndarrays

***`numpy.std`***

Equivalent function

`numpy.ma.`**`sum`**(*self*, *axis=None*, *dtype=None*, *out=None*, *keepdims=<no value>*) **`= <numpy.ma.core._frommethod object>`**

Return the sum of the array elements over the given axis.

Masked elements are set to 0 internally.

Refer to `numpy.sum` for full documentation.

**See also:**

***`numpy.ndarray.sum`***

corresponding function for ndarrays

***`numpy.sum`***

equivalent function

## Examples

```
>>> x = np.ma.array([[1,2,3],[4,5,6],[7,8,9]], mask=[0] + [1,0]*4)
>>> x
masked_array(
  data=[[1, --, 3],
        [--, 5, --],
        [7, --, 9]],
  mask=[[False,  True, False],
        [ True, False,  True],
        [False,  True, False]],
  fill_value=999999)
>>> x.sum()
25
>>> x.sum(axis=1)
masked_array(data=[4, 5, 16],
             mask=[False, False, False],
       fill_value=999999)
>>> x.sum(axis=0)
masked_array(data=[8, 5, 12],
             mask=[False, False, False],
       fill_value=999999)
>>> print(type(x.sum(axis=0, dtype=np.int64)[0]))
<class 'numpy.int64'>
```

`numpy.ma.`**`var`**(*self*, *axis=None*, *dtype=None*, *out=None*, *ddof=0*, *keepdims=<no value>*) **`= <numpy.ma.core._frommethod object>`**

Compute the variance along the specified axis.

Returns the variance of the array elements, a measure of the spread of a distribution. The variance is computed for the flattened array by default, otherwise over the specified axis.

**Parameters**

**a**

[array_like] Array containing numbers whose variance is desired. If *a* is not an array, a conversion is attempted.

**axis**

[None or int or tuple of ints, optional] Axis or axes along which the variance is computed. The default is to compute the variance of the flattened array.

New in version 1.7.0.

If this is a tuple of ints, a variance is performed over multiple axes, instead of a single axis or all the axes as before.

**dtype**

[data-type, optional] Type to use in computing the variance. For arrays of integer type the default is *float64*; for arrays of float types it is the same as the array type.

**out**

[ndarray, optional] Alternate output array in which to place the result. It must have the same shape as the expected output, but the type is cast if necessary.

**ddof**

[int, optional] "Delta Degrees of Freedom": the divisor used in the calculation is `N - ddof`, where `N` represents the number of elements. By default *ddof* is zero.

**keepdims**

[bool, optional] If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the input array.

If the default value is passed, then *keepdims* will not be passed through to the *`var`* method of sub-classes of *ndarray*, however any non-default value will be. If the sub-class' method does not implement *keepdims* any exceptions will be raised.

**Returns**

**variance**

[ndarray, see dtype parameter above] If `out=None`, returns a new array containing the variance; otherwise, a reference to the output array is returned.

**See also:**

*`std`*, *`mean`*, `nanmean`, `nanstd`, `nanvar`, `ufuncs-output-type`

## Notes

The variance is the average of the squared deviations from the mean, i.e., `var = mean(abs(x - x.mean())**2)`.

The mean is normally calculated as `x.sum() / N`, where `N = len(x)`. If, however, *ddof* is specified, the divisor `N - ddof` is used instead. In standard statistical practice, `ddof=1` provides an unbiased estimator of the variance of a hypothetical infinite population. `ddof=0` provides a maximum likelihood estimate of the variance for normally distributed variables.

Note that for complex numbers, the absolute value is taken before squaring, so that the result is always real and nonnegative.

For floating-point input, the variance is computed using the same precision the input has. Depending on the input data, this can cause the results to be inaccurate, especially for *float32* (see example below). Specifying a higher-accuracy accumulator using the `dtype` keyword can alleviate this issue.

## Examples

```
>>> a = np.array([[1, 2], [3, 4]])
>>> np.var(a)
1.25
>>> np.var(a, axis=0)
array([1.,  1.])
>>> np.var(a, axis=1)
array([0.25,  0.25])
```

In single precision, var() can be inaccurate:

```
>>> a = np.zeros((2, 512*512), dtype=np.float32)
>>> a[0, :] = 1.0
>>> a[1, :] = 0.1
```

(continues on next page)

(continued from previous page)

```
>>> np.var(a)
0.20250003
```

Computing the variance in float64 is more accurate:

```
>>> np.var(a, dtype=np.float64)
0.20249999932944759 # may vary
>>> ((1-0.55)**2 + (0.1-0.55)**2)/2
0.2025
```

## Minimum/maximum

| | |
|---|---|
| `ma.argmax`(self[, axis, fill_value, out]) | Returns array of indices of the maximum values along the given axis. |
| `ma.argmin`(self[, axis, fill_value, out]) | Return array of indices to the minimum values along the given axis. |
| `ma.max`(obj[, axis, out, fill_value, keepdims]) | Return the maximum along a given axis. |
| `ma.min`(obj[, axis, out, fill_value, keepdims]) | Return the minimum along a given axis. |
| `ma.ptp`(obj[, axis, out, fill_value, keepdims]) | Return (maximum - minimum) along the given dimension (i.e. |
| `ma.MaskedArray.argmax`(self[, axis, …]) | Returns array of indices of the maximum values along the given axis. |
| `ma.MaskedArray.argmin`(self[, axis, …]) | Return array of indices to the minimum values along the given axis. |
| `ma.MaskedArray.max`(self[, axis, out, …]) | Return the maximum along a given axis. |
| `ma.MaskedArray.min`(self[, axis, out, …]) | Return the minimum along a given axis. |
| `ma.MaskedArray.ptp`(self[, axis, out, …]) | Return (maximum - minimum) along the given dimension (i.e. |

`numpy.ma.`**`argmax`**(*self*, *axis=None*, *fill_value=None*, *out=None*) **`= <numpy.ma.core._frommethod object>`**

Returns array of indices of the maximum values along the given axis. Masked values are treated as if they had the value fill_value.

**Parameters**

**axis**

[{None, integer}] If None, the index is into the flattened array, otherwise along the specified axis

**fill_value**

[{var}, optional] Value used to fill in the masked values. If None, the output of maximum_fill_value(self._data) is used instead.

**out**

[{None, array}, optional] Array into which the result can be placed. Its type is preserved and it must be of the right shape to hold the output.

**Returns**

**index_array**

[{integer_array}]

### Examples

```
>>> a = np.arange(6).reshape(2,3)
>>> a.argmax()
5
>>> a.argmax(0)
array([1, 1, 1])
>>> a.argmax(1)
array([2, 2])
```

`numpy.ma.`**`argmin`**(*self*, *axis=None*, *fill_value=None*, *out=None*) **`= <numpy.ma.core._frommethod object>`**

Return array of indices to the minimum values along the given axis.

**Parameters**

**axis**

[{None, integer}] If None, the index is into the flattened array, otherwise along the specified axis

**fill_value**

[{var}, optional] Value used to fill in the masked values. If None, the output of minimum_fill_value(self._data) is used instead.

**out**

[{None, array}, optional] Array into which the result can be placed. Its type is preserved and it must be of the right shape to hold the output.

**Returns**

**ndarray or scalar**

If multi-dimension input, returns a new ndarray of indices to the minimum values along the given axis. Otherwise, returns a scalar of index to the minimum values along the given axis.

### Examples

```
>>> x = np.ma.array(np.arange(4), mask=[1,1,0,0])
>>> x.shape = (2,2)
>>> x
masked_array(
  data=[[--, --],
        [2, 3]],
  mask=[[ True,  True],
        [False, False]],
  fill_value=999999)
>>> x.argmin(axis=0, fill_value=-1)
array([0, 0])
>>> x.argmin(axis=0, fill_value=9)
array([1, 1])
```

`numpy.ma.`**`max`**(*obj*, *axis=None*, *out=None*, *fill_value=None*, *keepdims=<no value>*)

Return the maximum along a given axis.

**Parameters**

**axis**

[{None, int}, optional] Axis along which to operate. By default, `axis` is None and the flattened input is used.

**out**

[array_like, optional] Alternative output array in which to place the result. Must be of the same shape and buffer length as the expected output.

**fill_value**

[{var}, optional] Value used to fill in the masked values. If None, use the output of maximum_fill_value().

**keepdims**

[bool, optional] If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the array.

**Returns**

**amax**

[array_like] New array holding the result. If `out` was specified, `out` is returned.

**See also:**

***`maximum_fill_value`***

Returns the maximum filling value for a given datatype.

`numpy.ma.`**`min`**(*obj*, *axis=None*, *out=None*, *fill_value=None*, *keepdims=<no value>*)

Return the minimum along a given axis.

**Parameters**

**axis**

[{None, int}, optional] Axis along which to operate. By default, `axis` is None and the flattened input is used.

**out**

[array_like, optional] Alternative output array in which to place the result. Must be of the same shape and buffer length as the expected output.

**fill_value**

[{var}, optional] Value used to fill in the masked values. If None, use the output of `minimum_fill_value`.

**keepdims**

[bool, optional] If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the array.

**Returns**

**amin**

[array_like] New array holding the result. If `out` was specified, `out` is returned.

**See also:**

**`minimum_fill_value`**

Returns the minimum filling value for a given datatype.

`numpy.ma.`**`ptp`**`(`*obj*, *axis=None*, *out=None*, *fill_value=None*, *keepdims=<no value>*`)`

Return (maximum - minimum) along the given dimension (i.e. peak-to-peak value).

> **Warning:** *`ptp`* preserves the data type of the array. This means the return value for an input of signed integers with n bits (e.g. *np.int8*, *np.int16*, etc) is also a signed integer with n bits. In that case, peak-to-peak values greater than `2**(n-1)-1` will be returned as negative values. An example with a work-around is shown below.

**Parameters**

**axis**

[{None, int}, optional] Axis along which to find the peaks. If None (default) the flattened array is used.

**out**

[{None, array_like}, optional] Alternative output array in which to place the result. It must have the same shape and buffer length as the expected output but the type will be cast if necessary.

**fill_value**

[{var}, optional] Value used to fill in the masked values.

**keepdims**

[bool, optional] If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the array.

**Returns**

**ptp**

[ndarray.] A new array holding the result, unless `out` was specified, in which case a reference to `out` is returned.

## Examples

```
>>> x = np.ma.MaskedArray([[4, 9, 2, 10],
...                        [6, 9, 7, 12]])
```

```
>>> x.ptp(axis=1)
masked_array(data=[8, 6],
             mask=False,
       fill_value=999999)
```

```
>>> x.ptp(axis=0)
masked_array(data=[2, 0, 5, 2],
             mask=False,
       fill_value=999999)
```

```
>>> x.ptp()
10
```

This example shows that a negative value can be returned when the input is an array of signed integers.

```
>>> y = np.ma.MaskedArray([[1, 127],
...                        [0, 127],
...                        [-1, 127],
...                        [-2, 127]], dtype=np.int8)
>>> y.ptp(axis=1)
masked_array(data=[ 126,  127, -128, -127],
             mask=False,
       fill_value=999999,
            dtype=int8)
```

A work-around is to use the *view()* method to view the result as unsigned integers with the same bit width:

```
>>> y.ptp(axis=1).view(np.uint8)
masked_array(data=[126, 127, 128, 129],
             mask=False,
       fill_value=999999,
            dtype=uint8)
```

## Sorting

| | |
|---|---|
| `ma.argsort`(a[, axis, kind, order, endwith, …]) | Return an ndarray of indices that sort the array along the specified axis. |
| `ma.sort`(a[, axis, kind, order, endwith, …]) | Return a sorted copy of the masked array. |
| `ma.MaskedArray.argsort`(self[, axis, kind, …]) | Return an ndarray of indices that sort the array along the specified axis. |
| `ma.MaskedArray.sort`(self[, axis, kind, …]) | Sort the array, in-place |

`numpy.ma.`**`argsort`**(*a*, *axis=<no value>*, *kind=None*, *order=None*, *endwith=True*, *fill_value=None*)

Return an ndarray of indices that sort the array along the specified axis. Masked values are filled beforehand to *fill_value*.

**Parameters**

**axis**

[int, optional] Axis along which to sort. If None, the default, the flattened array is used.

Changed in version 1.13.0: Previously, the default was documented to be -1, but that was in error. At some future date, the default will change to -1, as originally intended. Until then, the axis should be given explicitly when `arr.ndim > 1`, to avoid a FutureWarning.

**kind**

[{‘quicksort’, ‘mergesort’, ‘heapsort’, ‘stable’}, optional] The sorting algorithm used.

**order**

[list, optional] When *a* is an array with fields defined, this argument specifies which fields to compare first, second, etc. Not all fields need be specified.

**endwith**

[{True, False}, optional] Whether missing values (if any) should be treated as the largest values (True) or the smallest values (False) When the array contains unmasked values at the same extremes of the datatype, the ordering of these values and the masked values is undefined.

**fill_value**

[{var}, optional] Value used internally for the masked values. If `fill_value` is not None, it supersedes `endwith`.

**Returns**

**index_array**

[ndarray, int] Array of indices that sort *a* along the specified axis. In other words, `a[index_array]` yields a sorted *a*.

**See also:**

***`MaskedArray.sort`***

Describes sorting algorithms used.

**`lexsort`**

Indirect stable sort with multiple keys.

***`numpy.ndarray.sort`***

Inplace sort.

## Notes

See *`sort`* for notes on the different sorting algorithms.

## Examples

```
>>> a = np.ma.array([3,2,1], mask=[False, False, True])
>>> a
masked_array(data=[3, 2, --],
             mask=[False, False,  True],
       fill_value=999999)
>>> a.argsort()
array([1, 0, 2])
```

`numpy.ma.`**`sort`**(*a*, *axis=-1*, *kind=None*, *order=None*, *endwith=True*, *fill_value=None*)

Return a sorted copy of the masked array.

Equivalent to creating a copy of the array and applying the MaskedArray `sort()` method.

Refer to `MaskedArray.sort` for the full documentation

**See also:**

**`MaskedArray.sort`**

equivalent method

### Algebra

| | |
|---|---|
| `ma.diag`(v[, k]) | Extract a diagonal or construct a diagonal array. |
| `ma.dot`(a, b[, strict, out]) | Return the dot product of two arrays. |
| `ma.identity`(n[, dtype]) | Return the identity array. |
| `ma.inner`(a, b) | Inner product of two arrays. |
| `ma.innerproduct`(a, b) | Inner product of two arrays. |
| `ma.outer`(a, b) | Compute the outer product of two vectors. |
| `ma.outerproduct`(a, b) | Compute the outer product of two vectors. |
| `ma.trace`(self[, offset, axis1, axis2, …]) | Return the sum along diagonals of the array. |
| `ma.transpose`(a[, axes]) | Permute the dimensions of an array. |
| `ma.MaskedArray.trace`([offset, axis1, axis2, …]) | Return the sum along diagonals of the array. |
| `ma.MaskedArray.transpose`(*axes) | Returns a view of the array with axes transposed. |

`numpy.ma.`**`diag`**(*v*, *k=0*)

Extract a diagonal or construct a diagonal array.

This function is the equivalent of `numpy.diag` that takes masked values into account, see `numpy.diag` for details.

**See also:**

**`numpy.diag`**

Equivalent function for ndarrays.

`numpy.ma.`**`dot`**(*a*, *b*, *strict=False*, *out=None*)

Return the dot product of two arrays.

This function is the equivalent of `numpy.dot` that takes masked values into account. Note that *strict* and *out* are in different position than in the method version. In order to maintain compatibility with the corresponding method, it is recommended that the optional arguments be treated as keyword only. At some point that may be mandatory.

**Note:** Works only with 2-D arrays at the moment.

**Parameters**

**a, b**

[masked_array_like] Inputs arrays.

**strict**

[bool, optional] Whether masked data are propagated (True) or set to 0 (False) for the computation. Default is False. Propagating the mask means that if a masked value appears in a row or column, the whole row or column is considered masked.

**out**

[masked_array, optional] Output argument. This must have the exact kind that would be returned if it was not used. In particular, it must have the right type, must be C-contiguous, and its dtype must be the dtype that would be returned for *dot(a,b)*. This is a performance feature.

Therefore, if these conditions are not met, an exception is raised, instead of attempting to be flexible.

New in version 1.10.2.

**See also:**

*`numpy.dot`*

Equivalent function for ndarrays.

## Examples

```
>>> a = np.ma.array([[1, 2, 3], [4, 5, 6]], mask=[[1, 0, 0], [0, 0, 0]])
>>> b = np.ma.array([[1, 2], [3, 4], [5, 6]], mask=[[1, 0], [0, 0], [0, 0]])
>>> np.ma.dot(a, b)
masked_array(
  data=[[21, 26],
        [45, 64]],
  mask=[[False, False],
        [False, False]],
  fill_value=999999)
>>> np.ma.dot(a, b, strict=True)
masked_array(
  data=[[--, --],
        [--, 64]],
  mask=[[ True,  True],
        [ True, False]],
  fill_value=999999)
```

`numpy.ma.`**`identity`**`(`*n*, *dtype=None*`)` **`= <numpy.ma.core._convert2ma object>`**

Return the identity array.

The identity array is a square array with ones on the main diagonal.

**Parameters**

**n**

[int] Number of rows (and columns) in *n* x *n* output.

**dtype**

[data-type, optional] Data-type of the output. Defaults to `float`.

**Returns**

**out**

[ndarray] *n* x *n* array with its main diagonal set to one, and all other elements 0.

## Examples

```
>>> np.identity(3)
array([[1.,  0.,  0.],
       [0.,  1.,  0.],
       [0.,  0.,  1.]])
```

`numpy.ma.`**`inner`**`(`*a*, *b*`)`

Inner product of two arrays.

Ordinary inner product of vectors for 1-D arrays (without complex conjugation), in higher dimensions a sum product over the last axes.

**Parameters**

**a, b**

[array_like] If *a* and *b* are nonscalar, their last dimensions must match.

**Returns**

**out**

[ndarray] *out.shape = a.shape[:-1] + b.shape[:-1]*

**Raises**

**ValueError**

If the last dimension of *a* and *b* has different size.

**See also:**

**`tensordot`**

Sum products over arbitrary axes.

*`dot`*

Generalised matrix product, using second last dimension of *b*.

**`einsum`**

Einstein summation convention.

## Notes

Masked values are replaced by 0.

For vectors (1-D arrays) it computes the ordinary inner-product:

```
np.inner(a, b) = sum(a[:]*b[:])
```

More generally, if *ndim(a) = r > 0* and *ndim(b) = s > 0*:

```
np.inner(a, b) = np.tensordot(a, b, axes=(-1,-1))
```

or explicitly:

```
np.inner(a, b)[i0,...,ir-1,j0,...,js-1]
     = sum(a[i0,...,ir-1,:]*b[j0,...,js-1,:])
```

In addition *a* or *b* may be scalars, in which case:

```
np.inner(a,b) = a*b
```

### Examples

Ordinary inner product for vectors:

```
>>> a = np.array([1,2,3])
>>> b = np.array([0,1,0])
>>> np.inner(a, b)
2
```

A multidimensional example:

```
>>> a = np.arange(24).reshape((2,3,4))
>>> b = np.arange(4)
>>> np.inner(a, b)
array([[ 14,  38,  62],
       [ 86, 110, 134]])
```

An example where *b* is a scalar:

```
>>> np.inner(np.eye(2), 7)
array([[7., 0.],
       [0., 7.]])
```

`numpy.ma.`**`innerproduct`**`(`*a*, *b*`)`

Inner product of two arrays.

Ordinary inner product of vectors for 1-D arrays (without complex conjugation), in higher dimensions a sum product over the last axes.

**Parameters**

**a, b**

[array_like] If *a* and *b* are nonscalar, their last dimensions must match.

**Returns**

**out**

[ndarray] *out.shape = a.shape[:-1] + b.shape[:-1]*

**Raises**

**ValueError**

If the last dimension of *a* and *b* has different size.

**See also:**

**`tensordot`**

Sum products over arbitrary axes.

***`dot`***

Generalised matrix product, using second last dimension of *b*.

**`einsum`**

Einstein summation convention.

## Notes

Masked values are replaced by 0.

For vectors (1-D arrays) it computes the ordinary inner-product:

```
np.inner(a, b) = sum(a[:]*b[:])
```

More generally, if *ndim(a)* = *r* > *0* and *ndim(b)* = *s* > *0*:

```
np.inner(a, b) = np.tensordot(a, b, axes=(-1,-1))
```

or explicitly:

```
np.inner(a, b)[i0,...,ir-1,j0,...,js-1]
     = sum(a[i0,...,ir-1,:]*b[j0,...,js-1,:])
```

In addition *a* or *b* may be scalars, in which case:

```
np.inner(a,b) = a*b
```

## Examples

Ordinary inner product for vectors:

```
>>> a = np.array([1,2,3])
>>> b = np.array([0,1,0])
>>> np.inner(a, b)
2
```

A multidimensional example:

```
>>> a = np.arange(24).reshape((2,3,4))
>>> b = np.arange(4)
>>> np.inner(a, b)
array([[ 14,  38,  62],
       [ 86, 110, 134]])
```

An example where *b* is a scalar:

```
>>> np.inner(np.eye(2), 7)
array([[7., 0.],
       [0., 7.]])
```

`numpy.ma.`**`outer`**(*a*, *b*)

Compute the outer product of two vectors.

Given two vectors, `a = [a0, a1, ..., aM]` and `b = [b0, b1, ..., bN]`, the outer product [1] is:

```
[[a0*b0  a0*b1 ... a0*bN ]
 [a1*b0    .
 [ ...          .
 [aM*b0            aM*bN ]]
```

**Parameters**

**a**

[(M,) array_like] First input vector. Input is flattened if not already 1-dimensional.

**b**

[(N,) array_like] Second input vector. Input is flattened if not already 1-dimensional.

**out**

[(M, N) ndarray, optional] A location where the result is stored

New in version 1.9.0.

**Returns**

**out**

[(M, N) ndarray] `out[i, j] = a[i] * b[j]`

**See also:**

*`inner`*

**`einsum`**

`einsum('i,j->ij', a.ravel(), b.ravel())` is the equivalent.

**`ufunc.outer`**

A generalization to dimensions other than 1D and other operations. `np.multiply.outer(a.ravel(), b.ravel())` is the equivalent.

**`tensordot`**

`np.tensordot(a.ravel(), b.ravel(), axes=((), ()))` is the equivalent.

## Notes

Masked values are replaced by 0.

### References

[1]

### Examples

Make a (*very* coarse) grid for computing a Mandelbrot set:

```
>>> rl = np.outer(np.ones((5,)), np.linspace(-2, 2, 5))
>>> rl
array([[-2., -1.,  0.,  1.,  2.],
       [-2., -1.,  0.,  1.,  2.],
       [-2., -1.,  0.,  1.,  2.],
       [-2., -1.,  0.,  1.,  2.],
       [-2., -1.,  0.,  1.,  2.]])
>>> im = np.outer(1j*np.linspace(2, -2, 5), np.ones((5,)))
>>> im
array([[0.+2.j, 0.+2.j, 0.+2.j, 0.+2.j, 0.+2.j],
       [0.+1.j, 0.+1.j, 0.+1.j, 0.+1.j, 0.+1.j],
       [0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j],
       [0.-1.j, 0.-1.j, 0.-1.j, 0.-1.j, 0.-1.j],
       [0.-2.j, 0.-2.j, 0.-2.j, 0.-2.j, 0.-2.j]])
>>> grid = rl + im
>>> grid
array([[-2.+2.j, -1.+2.j,  0.+2.j,  1.+2.j,  2.+2.j],
       [-2.+1.j, -1.+1.j,  0.+1.j,  1.+1.j,  2.+1.j],
       [-2.+0.j, -1.+0.j,  0.+0.j,  1.+0.j,  2.+0.j],
       [-2.-1.j, -1.-1.j,  0.-1.j,  1.-1.j,  2.-1.j],
       [-2.-2.j, -1.-2.j,  0.-2.j,  1.-2.j,  2.-2.j]])
```

An example using a "vector" of letters:

```
>>> x = np.array(['a', 'b', 'c'], dtype=object)
>>> np.outer(x, [1, 2, 3])
array([['a', 'aa', 'aaa'],
       ['b', 'bb', 'bbb'],
       ['c', 'cc', 'ccc']], dtype=object)
```

`numpy.ma.`**`outerproduct`**(*a*, *b*)

Compute the outer product of two vectors.

Given two vectors, `a = [a0, a1, ..., aM]` and `b = [b0, b1, ..., bN]`, the outer product [1] is:

```
[[a0*b0  a0*b1 ... a0*bN ]
 [a1*b0    .
 [ ...          .
 [aM*b0            aM*bN ]]
```

**Parameters**

**a**

[(M,) array_like] First input vector. Input is flattened if not already 1-dimensional.

**b**

[(N,) array_like] Second input vector. Input is flattened if not already 1-dimensional.

**out**

[(M, N) ndarray, optional] A location where the result is stored

New in version 1.9.0.

**Returns**

**out**

[(M, N) ndarray] `out[i, j] = a[i] * b[j]`

**See also:**

*`inner`*

**`einsum`**

`einsum('i,j->ij', a.ravel(), b.ravel())` is the equivalent.

**`ufunc.outer`**

A generalization to dimensions other than 1D and other operations. `np.multiply.outer(a.ravel(), b.ravel())` is the equivalent.

**`tensordot`**

`np.tensordot(a.ravel(), b.ravel(), axes=((), ()))` is the equivalent.

## Notes

Masked values are replaced by 0.

## References

[1]

## Examples

Make a (*very* coarse) grid for computing a Mandelbrot set:

```
>>> rl = np.outer(np.ones((5,)), np.linspace(-2, 2, 5))
>>> rl
array([[-2., -1.,  0.,  1.,  2.],
       [-2., -1.,  0.,  1.,  2.],
       [-2., -1.,  0.,  1.,  2.],
       [-2., -1.,  0.,  1.,  2.],
       [-2., -1.,  0.,  1.,  2.]])
>>> im = np.outer(1j*np.linspace(2, -2, 5), np.ones((5,)))
>>> im
array([[0.+2.j, 0.+2.j, 0.+2.j, 0.+2.j, 0.+2.j],
       [0.+1.j, 0.+1.j, 0.+1.j, 0.+1.j, 0.+1.j],
       [0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j],
       [0.-1.j, 0.-1.j, 0.-1.j, 0.-1.j, 0.-1.j],
       [0.-2.j, 0.-2.j, 0.-2.j, 0.-2.j, 0.-2.j]])
>>> grid = rl + im
>>> grid
```

(continues on next page)

(continued from previous page)

```
array([[-2.+2.j, -1.+2.j,  0.+2.j,  1.+2.j,  2.+2.j],
       [-2.+1.j, -1.+1.j,  0.+1.j,  1.+1.j,  2.+1.j],
       [-2.+0.j, -1.+0.j,  0.+0.j,  1.+0.j,  2.+0.j],
       [-2.-1.j, -1.-1.j,  0.-1.j,  1.-1.j,  2.-1.j],
       [-2.-2.j, -1.-2.j,  0.-2.j,  1.-2.j,  2.-2.j]])
```

An example using a "vector" of letters:

```
>>> x = np.array(['a', 'b', 'c'], dtype=object)
>>> np.outer(x, [1, 2, 3])
array([['a', 'aa', 'aaa'],
       ['b', 'bb', 'bbb'],
       ['c', 'cc', 'ccc']], dtype=object)
```

`numpy.ma.`**`trace`**(*self*, *offset=0*, *axis1=0*, *axis2=1*, *dtype=None*, *out=None*) *a.trace(offset=0*, *axis1=0*, *axis2=1*, *dtype=None*, *out=None*) **`= <numpy.ma.core._frommethod object>`**

Return the sum along diagonals of the array.

Refer to `numpy.trace` for full documentation.

**See also:**

**`numpy.trace`**

equivalent function

## Polynomial fit

| | |
|---|---|
| `ma.vander`(x[, n]) | Generate a Vandermonde matrix. |
| `ma.polyfit`(x, y, deg[, rcond, full, w, cov]) | Least squares polynomial fit. |

`numpy.ma.`**`vander`**(*x*, *n=None*)

Generate a Vandermonde matrix.

The columns of the output matrix are powers of the input vector. The order of the powers is determined by the *increasing* boolean argument. Specifically, when *increasing* is False, the *i*-th output column is the input vector raised element-wise to the power of `N - i - 1`. Such a matrix with a geometric progression in each row is named for Alexandre- Theophile Vandermonde.

**Parameters**

**x**

[array_like] 1-D input array.

**N**

[int, optional] Number of columns in the output. If *N* is not specified, a square array is returned (`N = len(x)`).

**increasing**

[bool, optional] Order of the powers of the columns. If True, the powers increase from left to right, if False (the default) they are reversed.

New in version 1.9.0.

**Returns**

**out**

[ndarray] Vandermonde matrix. If *increasing* is False, the first column is `x^(N-1)`, the second `x^(N-2)` and so forth. If *increasing* is True, the columns are `x^0, x^1, ..., x^(N-1)`.

**See also:**

`polynomial.polynomial.polyvander`

### Notes

Masked values in the input array result in rows of zeros.

### Examples

```
>>> x = np.array([1, 2, 3, 5])
>>> N = 3
>>> np.vander(x, N)
array([[ 1,  1,  1],
       [ 4,  2,  1],
       [ 9,  3,  1],
       [25,  5,  1]])
```

```
>>> np.column_stack([x**(N-1-i) for i in range(N)])
array([[ 1,  1,  1],
       [ 4,  2,  1],
       [ 9,  3,  1],
       [25,  5,  1]])
```

```
>>> x = np.array([1, 2, 3, 5])
>>> np.vander(x)
array([[  1,   1,   1,   1],
       [  8,   4,   2,   1],
       [ 27,   9,   3,   1],
       [125,  25,   5,   1]])
>>> np.vander(x, increasing=True)
array([[  1,   1,   1,   1],
       [  1,   2,   4,   8],
       [  1,   3,   9,  27],
       [  1,   5,  25, 125]])
```

The determinant of a square Vandermonde matrix is the product of the differences between the values of the input vector:

```
>>> np.linalg.det(np.vander(x))
48.000000000000043 # may vary
>>> (5-3)*(5-2)*(5-1)*(3-2)*(3-1)*(2-1)
48
```

`numpy.ma.`**`polyfit`**(*x*, *y*, *deg*, *rcond=None*, *full=False*, *w=None*, *cov=False*)

Least squares polynomial fit.

Fit a polynomial `p(x) = p[0] * x**deg + ... + p[deg]` of degree *deg* to points *(x, y)*. Returns a vector of coefficients *p* that minimises the squared error in the order *deg*, *deg-1*, … *0*.

The `Polynomial.fit` class method is recommended for new code as it is more stable numerically. See the documentation of the method for more information.

**Parameters**

**x**

[array_like, shape (M,)] x-coordinates of the M sample points `(x[i], y[i])`.

**y**

[array_like, shape (M,) or (M, K)] y-coordinates of the sample points. Several data sets of sample points sharing the same x-coordinates can be fitted at once by passing in a 2D-array that contains one dataset per column.

**deg**

[int] Degree of the fitting polynomial

**rcond**

[float, optional] Relative condition number of the fit. Singular values smaller than this relative to the largest singular value will be ignored. The default value is len(x)*eps, where eps is the relative precision of the float type, about 2e-16 in most cases.

**full**

[bool, optional] Switch determining nature of return value. When it is False (the default) just the coefficients are returned, when True diagnostic information from the singular value decomposition is also returned.

**w**

[array_like, shape (M,), optional] Weights to apply to the y-coordinates of the sample points. For gaussian uncertainties, use 1/sigma (not 1/sigma**2).

**cov**

[bool or str, optional] If given and not *False*, return not just the estimate but also its covariance matrix. By default, the covariance are scaled by chi2/sqrt(N-dof), i.e., the weights are presumed to be unreliable except in a relative sense and everything is scaled such that the reduced chi2 is unity. This scaling is omitted if `cov='unscaled'`, as is relevant for the case that the weights are 1/sigma**2, with sigma known to be a reliable estimate of the uncertainty.

**Returns**

**p**

[ndarray, shape (deg + 1,) or (deg + 1, K)] Polynomial coefficients, highest power first. If *y* was 2-D, the coefficients for *k*-th data set are in `p[:,k]`.

**residuals, rank, singular_values, rcond**

Present only if *full* = True. Residuals is sum of squared residuals of the least-squares fit, the effective rank of the scaled Vandermonde coefficient matrix, its singular values, and the specified value of *rcond*. For more details, see *linalg.lstsq*.

**V**

[ndarray, shape (M,M) or (M,M,K)] Present only if *full* = False and *cov'=True. The covariance matrix of the polynomial coefficient estimates. The diagonal of this matrix are the variance* *estimates for each coefficient. If y is a 2-D array, then the covariance matrix for the 'k*-th data set are in `V[:,:,k]`

**Warns**

**RankWarning**

The rank of the coefficient matrix in the least-squares fit is deficient. The warning is only raised if *full* = False.

The warnings can be turned off by

```
>>> import warnings
>>> warnings.simplefilter('ignore', np.RankWarning)
```

**See also:**

**`polyval`**

Compute polynomial values.

**`linalg.lstsq`**

Computes a least-squares fit.

**`scipy.interpolate.UnivariateSpline`**

Computes spline fits.

## Notes

Any masked values in x is propagated in y, and vice-versa.

The solution minimizes the squared error

$$E = \sum_{j=0}^{k} |p(x_j) - y_j|^2$$

in the equations:

```
x[0]**n * p[0] + ... + x[0] * p[n-1] + p[n] = y[0]
x[1]**n * p[0] + ... + x[1] * p[n-1] + p[n] = y[1]
...
x[k]**n * p[0] + ... + x[k] * p[n-1] + p[n] = y[k]
```

The coefficient matrix of the coefficients *p* is a Vandermonde matrix.

`polyfit` issues a *RankWarning* when the least-squares fit is badly conditioned. This implies that the best fit is not well-defined due to numerical error. The results may be improved by lowering the polynomial degree or by replacing *x* by *x* - *x*.mean(). The *rcond* parameter can also be set to a value smaller than its default, but the resulting fit may be spurious: including contributions from the small singular values can add numerical noise to the result.

Note that fitting polynomial coefficients is inherently badly conditioned when the degree of the polynomial is large or the interval of sample points is badly centered. The quality of the fit should always be checked in these cases. When polynomial fits are not satisfactory, splines may be a good alternative.

### References

[1], [2]

### Examples

```
>>> import warnings
>>> x = np.array([0.0, 1.0, 2.0, 3.0,  4.0,  5.0])
>>> y = np.array([0.0, 0.8, 0.9, 0.1, -0.8, -1.0])
>>> z = np.polyfit(x, y, 3)
>>> z
array([ 0.08703704, -0.81349206,  1.69312169, -0.03968254]) # may vary
```

It is convenient to use *poly1d* objects for dealing with polynomials:

```
>>> p = np.poly1d(z)
>>> p(0.5)
0.6143849206349179 # may vary
>>> p(3.5)
-0.34732142857143039 # may vary
>>> p(10)
22.579365079365115 # may vary
```

High-order polynomials may oscillate wildly:

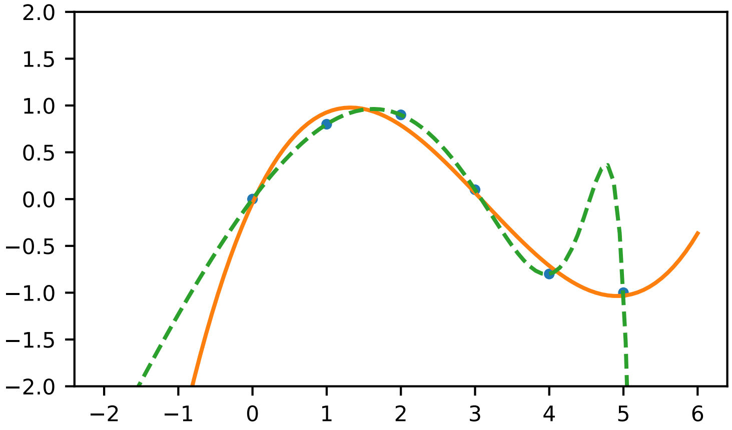
```
>>> with warnings.catch_warnings():
...     warnings.simplefilter('ignore', np.RankWarning)
...     p30 = np.poly1d(np.polyfit(x, y, 30))
...
>>> p30(4)
-0.80000000000000204 # may vary
>>> p30(5)
-0.99999999999999445 # may vary
>>> p30(4.5)
-0.10547061179440398 # may vary
```

Illustration:

```
>>> import matplotlib.pyplot as plt
>>> xp = np.linspace(-2, 6, 100)
>>> _ = plt.plot(x, y, '.', xp, p(xp), '-', xp, p30(xp), '--')
>>> plt.ylim(-2,2)
(-2, 2)
>>> plt.show()
```

## Clipping and rounding

| | |
|---|---|
| *`ma.around`*(a, *args, **kwargs) | Round an array to the given number of decimals. |
| *`ma.clip`*(a, a_min, a_max[, out]) | Clip (limit) the values in an array. |
| *`ma.round`*(a[, decimals, out]) | Return a copy of a, rounded to 'decimals' places. |
| *`ma.MaskedArray.clip`*([min, max, out]) | Return an array whose values are limited to `[min, max]`. |
| *`ma.MaskedArray.round`*(self[, decimals, out]) | Return each element rounded to the given number of decimals. |



`numpy.ma.`**`around`**(*a*, **args*, ***kwargs*) **`= <numpy.ma.core._MaskedUnaryOperation object>`**

Round an array to the given number of decimals.

**See also:**

***`around`***

equivalent function; see for details.

`numpy.ma.`**`clip`**(*a*, *a_min*, *a_max*, *out=None*, ***kwargs*)

Clip (limit) the values in an array.

Given an interval, values outside the interval are clipped to the interval edges. For example, if an interval of `[0, 1]` is specified, values smaller than 0 become 0, and values larger than 1 become 1.

Equivalent to but faster than `np.minimum(a_max, np.maximum(a, a_min))`.

No check is performed to ensure `a_min < a_max`.

**Parameters**

**a**

[array_like] Array containing elements to clip.

**a_min**

[scalar or array_like or None] Minimum value. If None, clipping is not performed on lower interval edge. Not more than one of *a_min* and *a_max* may be None.

**a_max**

[scalar or array_like or None] Maximum value. If None, clipping is not performed on upper interval edge. Not more than one of *a_min* and *a_max* may be None. If *a_min* or *a_max* are array_like, then the three arrays will be broadcasted to match their shapes.

**out**

[ndarray, optional] The results will be placed in this array. It may be the input array for in-place clipping. *out* must be of the right shape to hold the output. Its type is preserved.

****kwargs**

For other keyword-only arguments, see the *ufunc docs*.

New in version 1.17.0.

**Returns**

**clipped_array**

[ndarray] An array with the elements of *a*, but where values < *a_min* are replaced with *a_min*, and those > *a_max* with *a_max*.

**See also:**

`ufuncs-output-type`

## Examples

```
>>> a = np.arange(10)
>>> np.clip(a, 1, 8)
array([1, 1, 2, 3, 4, 5, 6, 7, 8, 8])
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> np.clip(a, 3, 6, out=a)
array([3, 3, 3, 3, 4, 5, 6, 6, 6, 6])
>>> a = np.arange(10)
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> np.clip(a, [3, 4, 1, 1, 1, 4, 4, 4, 4, 4], 8)
array([3, 4, 2, 3, 4, 5, 6, 7, 8, 8])
```

`numpy.ma.`**`round`**`(`*a*, *decimals=0*, *out=None*`)`

Return a copy of a, rounded to 'decimals' places.

When 'decimals' is negative, it specifies the number of positions to the left of the decimal point. The real and imaginary parts of complex numbers are rounded separately. Nothing is done if the array is not of float type and 'decimals' is greater than or equal to 0.

**Parameters**

**decimals**

[int] Number of decimals to round to. May be negative.

**out**

[array_like] Existing array to use for output. If not given, returns a default copy of a.

## Notes

If out is given and does not have a mask attribute, the mask of a is lost!

### Miscellanea

| | |
|---|---|
| `ma.allequal`(a, b[, fill_value]) | Return True if all entries of a and b are equal, using fill_value as a truth value where either or both are masked. |
| `ma.allclose`(a, b[, masked_equal, rtol, atol]) | Returns True if two arrays are element-wise equal within a tolerance. |
| `ma.apply_along_axis`(func1d, axis, arr, …) | Apply a function to 1-D slices along the given axis. |
| `ma.apply_over_axes`(func, a, axes) | Apply a function repeatedly over multiple axes. |
| `ma.arange`([start,] stop[, step,][, dtype]) | Return evenly spaced values within a given interval. |
| `ma.choose`(indices, choices[, out, mode]) | Use an index array to construct a new array from a set of choices. |
| `ma.ediff1d`(arr[, to_end, to_begin]) | Compute the differences between consecutive elements of an array. |
| `ma.indices`(dimensions[, dtype, sparse]) | Return an array representing the indices of a grid. |
| `ma.where`(condition[, x, y]) | Return a masked array with elements from *x* or *y*, depending on condition. |

`numpy.ma.`**`allequal`**(*a*, *b*, *fill_value=True*)

Return True if all entries of a and b are equal, using fill_value as a truth value where either or both are masked.

**Parameters**

**a, b**

[array_like] Input arrays to compare.

**fill_value**

[bool, optional] Whether masked values in a or b are considered equal (True) or not (False).

**Returns**

**y**

[bool] Returns True if the two arrays are equal within the given tolerance, False otherwise. If either array contains NaN, then False is returned.

**See also:**

*`all`*, *`any`*, *`numpy.ma.allclose`*

## Examples

```
>>> a = np.ma.array([1e10, 1e-7, 42.0], mask=[0, 0, 1])
>>> a
masked_array(data=[10000000000.0, 1e-07, --],
             mask=[False, False,  True],
       fill_value=1e+20)
```

```
>>> b = np.array([1e10, 1e-7, -42.0])
>>> b
array([  1.00000000e+10,   1.00000000e-07,  -4.20000000e+01])
>>> np.ma.allequal(a, b, fill_value=False)
False
>>> np.ma.allequal(a, b)
True
```

`numpy.ma.`**`allclose`**(*a*, *b*, *masked_equal=True*, *rtol=1e-05*, *atol=1e-08*)

Returns True if two arrays are element-wise equal within a tolerance.

This function is equivalent to *`allclose`* except that masked values are treated as equal (default) or unequal, depending on the *`masked_equal`* argument.

**Parameters**

**a, b**

[array_like] Input arrays to compare.

**masked_equal**

[bool, optional] Whether masked values in *a* and *b* are considered equal (True) or not (False). They are considered equal by default.

**rtol**

[float, optional] Relative tolerance. The relative difference is equal to `rtol * b`. Default is 1e-5.

**atol**

[float, optional] Absolute tolerance. The absolute difference is equal to *atol*. Default is 1e-8.

**Returns**

**y**

[bool] Returns True if the two arrays are equal within the given tolerance, False otherwise. If either array contains NaN, then False is returned.

**See also:**

*`all`*, *`any`*

***`numpy.allclose`***

the non-masked *`allclose`*.

## Notes

If the following equation is element-wise True, then *`allclose`* returns True:

```
absolute(`a` - `b`) <= (`atol` + `rtol` * absolute(`b`))
```

Return True if all elements of *a* and *b* are equal subject to given tolerances.

### Examples

```
>>> a = np.ma.array([1e10, 1e-7, 42.0], mask=[0, 0, 1])
>>> a
masked_array(data=[10000000000.0, 1e-07, --],
             mask=[False, False,  True],
       fill_value=1e+20)
>>> b = np.ma.array([1e10, 1e-8, -42.0], mask=[0, 0, 1])
>>> np.ma.allclose(a, b)
False
```

```
>>> a = np.ma.array([1e10, 1e-8, 42.0], mask=[0, 0, 1])
>>> b = np.ma.array([1.00001e10, 1e-9, -42.0], mask=[0, 0, 1])
>>> np.ma.allclose(a, b)
True
>>> np.ma.allclose(a, b, masked_equal=False)
False
```

Masked values are not compared directly.

```
>>> a = np.ma.array([1e10, 1e-8, 42.0], mask=[0, 0, 1])
>>> b = np.ma.array([1.00001e10, 1e-9, 42.0], mask=[0, 0, 1])
>>> np.ma.allclose(a, b)
True
>>> np.ma.allclose(a, b, masked_equal=False)
False
```

`numpy.ma.`**`apply_along_axis`**(*func1d*, *axis*, *arr*, **args*, ***kwargs*)

Apply a function to 1-D slices along the given axis.

Execute *func1d(a, *args, **kwargs)* where *func1d* operates on 1-D arrays and *a* is a 1-D slice of *arr* along *axis*.

This is equivalent to (but faster than) the following use of *ndindex* and *s_*, which sets each of `ii`, `jj`, and `kk` to a tuple of indices:

```
Ni, Nk = a.shape[:axis], a.shape[axis+1:]
for ii in ndindex(Ni):
    for kk in ndindex(Nk):
        f = func1d(arr[ii + s_[:,] + kk])
        Nj = f.shape
        for jj in ndindex(Nj):
            out[ii + jj + kk] = f[jj]
```

Equivalently, eliminating the inner loop, this can be expressed as:

```
Ni, Nk = a.shape[:axis], a.shape[axis+1:]
for ii in ndindex(Ni):
    for kk in ndindex(Nk):
        out[ii + s_[...,] + kk] = func1d(arr[ii + s_[:,] + kk])
```

**Parameters**

**func1d**

[function (M,) -> (Nj…)] This function should accept 1-D arrays. It is applied to 1-D slices of *arr* along the specified axis.

**axis**

[integer] Axis along which *arr* is sliced.

**arr**

[ndarray (Ni…, M, Nk…)] Input array.

**args**

[any] Additional arguments to *func1d*.

**kwargs**

[any] Additional named arguments to *func1d*.

New in version 1.9.0.

**Returns**

**out**

[ndarray (Ni…, Nj…, Nk…)] The output array. The shape of *out* is identical to the shape of *arr*, except along the *axis* dimension. This axis is removed, and replaced with new dimensions equal to the shape of the return value of *func1d*. So if *func1d* returns a scalar *out* will have one fewer dimensions than *arr*.

**See also:**

***apply_over_axes***

Apply a function repeatedly over multiple axes.

### Examples

```
>>> def my_func(a):
...     """Average first and last element of a 1-D array"""
...     return (a[0] + a[-1]) * 0.5
>>> b = np.array([[1,2,3], [4,5,6], [7,8,9]])
>>> np.apply_along_axis(my_func, 0, b)
array([4., 5., 6.])
>>> np.apply_along_axis(my_func, 1, b)
array([2.,  5.,  8.])
```

For a function that returns a 1D array, the number of dimensions in *outarr* is the same as *arr*.

```
>>> b = np.array([[8,1,7], [4,3,9], [5,2,6]])
>>> np.apply_along_axis(sorted, 1, b)
array([[1, 7, 8],
       [3, 4, 9],
       [2, 5, 6]])
```

For a function that returns a higher dimensional array, those dimensions are inserted in place of the *axis* dimension.

```
>>> b = np.array([[1,2,3], [4,5,6], [7,8,9]])
>>> np.apply_along_axis(np.diag, -1, b)
array([[[1, 0, 0],
        [0, 2, 0],
        [0, 0, 3]],
       [[4, 0, 0],
        [0, 5, 0],
```

(continues on next page)

(continued from previous page)

```
        [0, 0, 6]],
       [[7, 0, 0],
        [0, 8, 0],
        [0, 0, 9]]])
```

`numpy.ma.`**`apply_over_axes`**(*func*, *a*, *axes*)

Apply a function repeatedly over multiple axes.

*func* is called as *res = func(a, axis)*, where *axis* is the first element of *axes*. The result *res* of the function call must have either the same dimensions as *a* or one less dimension. If *res* has one less dimension than *a*, a dimension is inserted before *axis*. The call to *func* is then repeated for each axis in *axes*, with *res* as the first argument.

**Parameters**

**func**

[function] This function must take two arguments, *func(a, axis)*.

**a**

[array_like] Input array.

**axes**

[array_like] Axes over which *func* is applied; the elements must be integers.

**Returns**

**apply_over_axis**

[ndarray] The output array. The number of dimensions is the same as *a*, but the shape can be different. This depends on whether *func* changes the shape of its output with respect to its input.

**See also:**

***`apply_along_axis`***

Apply a function to 1-D slices of an array along the given axis.

## Examples

```
>>> a = np.ma.arange(24).reshape(2,3,4)
>>> a[:,0,1] = np.ma.masked
>>> a[:,1,:] = np.ma.masked
>>> a
masked_array(
  data=[[[0, --, 2, 3],
         [--, --, --, --],
         [8, 9, 10, 11]],
        [[12, --, 14, 15],
         [--, --, --, --],
         [20, 21, 22, 23]]],
  mask=[[[False,  True, False, False],
         [ True,  True,  True,  True],
         [False, False, False, False]],
        [[False,  True, False, False],
```

(continues on next page)

(continued from previous page)

```
         [ True,  True,  True,  True],
         [False, False, False, False]]],
  fill_value=999999)
>>> np.ma.apply_over_axes(np.ma.sum, a, [0,2])
masked_array(
  data=[[[46],
         [--],
         [124]]],
  mask=[[[False],
         [ True],
         [False]]],
  fill_value=999999)
```

Tuple axis arguments to ufuncs are equivalent:

```
>>> np.ma.sum(a, axis=(0,2)).reshape((1,-1,1))
masked_array(
  data=[[[46],
         [--],
         [124]]],
  mask=[[[False],
         [ True],
         [False]]],
  fill_value=999999)
```

`numpy.ma.`**`arange`**`(`[*start*], *stop*[, *step*], *dtype=None*`)` **`= <numpy.ma.core._convert2ma object>`**

Return evenly spaced values within a given interval.

Values are generated within the half-open interval `[start, stop)` (in other words, the interval including *start* but excluding *stop*). For integer arguments the function is equivalent to the Python built-in *range* function, but returns an ndarray rather than a list.

When using a non-integer step, such as 0.1, the results will often not be consistent. It is better to use `numpy.linspace` for these cases.

**Parameters**

**start**

[number, optional] Start of interval. The interval includes this value. The default start value is 0.

**stop**

[number] End of interval. The interval does not include this value, except in some cases where *step* is not an integer and floating point round-off affects the length of *out*.

**step**

[number, optional] Spacing between values. For any output *out*, this is the distance between two adjacent values, `out[i+1] - out[i]`. The default step size is 1. If *step* is specified as a position argument, *start* must also be given.

**dtype**

[dtype] The type of the output array. If *dtype* is not given, infer the data type from the other input arguments.

**Returns**

**arange**

[ndarray] Array of evenly spaced values.

For floating point arguments, the length of the result is `ceil((stop - start)/step)`. Because of floating point overflow, this rule may result in the last element of *out* being greater than *stop*.

**See also:**

***`numpy.linspace`***

Evenly spaced numbers with careful handling of endpoints.

***`numpy.ogrid`***

Arrays of evenly spaced numbers in N-dimensions.

***`numpy.mgrid`***

Grid-shaped arrays of evenly spaced numbers in N-dimensions.

## Examples

```
>>> np.arange(3)
array([0, 1, 2])
>>> np.arange(3.0)
array([ 0.,  1.,  2.])
>>> np.arange(3,7)
array([3, 4, 5, 6])
>>> np.arange(3,7,2)
array([3, 5])
```

`numpy.ma.`**`choose`**(*indices*, *choices*, *out=None*, *mode='raise'*)

Use an index array to construct a new array from a set of choices.

Given an array of integers and a set of n choice arrays, this method will create a new array that merges each of the choice arrays. Where a value in *a* is i, the new array will have the value that choices[i] contains in the same place.

**Parameters**

**a**

[ndarray of ints] This array must contain integers in `[0, n-1]`, where n is the number of choices.

**choices**

[sequence of arrays] Choice arrays. The index array and all of the choices should be broadcastable to the same shape.

**out**

[array, optional] If provided, the result will be inserted into this array. It should be of the appropriate shape and *dtype*.

**mode**

[{'raise', 'wrap', 'clip'}, optional] Specifies how out-of-bounds indices will behave.

- 'raise' : raise an error

- 'wrap' : wrap around
- 'clip' : clip to the range

**Returns**

**merged_array**

[array]

**See also:**

***choose***

equivalent function

### Examples

```
>>> choice = np.array([[1,1,1], [2,2,2], [3,3,3]])
>>> a = np.array([2, 1, 0])
>>> np.ma.choose(a, choice)
masked_array(data=[3, 2, 1],
             mask=False,
       fill_value=999999)
```

`numpy.ma.`**`ediff1d`**(*arr*, *to_end=None*, *to_begin=None*)

Compute the differences between consecutive elements of an array.

This function is the equivalent of *`numpy.ediff1d`* that takes masked values into account, see *`numpy.ediff1d`* for details.

**See also:**

***numpy.ediff1d***

Equivalent function for ndarrays.

`numpy.ma.`**`indices`**(*dimensions*, *dtype=<class 'int'>*, *sparse=False*)

Return an array representing the indices of a grid.

Compute an array where the subarrays contain index values 0, 1, … varying only along the corresponding axis.

**Parameters**

**dimensions**

[sequence of ints] The shape of the grid.

**dtype**

[dtype, optional] Data type of the result.

**sparse**

[boolean, optional] Return a sparse representation of the grid instead of a dense representation. Default is False.

New in version 1.17.

**Returns**

**grid**

[one ndarray or tuple of ndarrays]

**If sparse is False:**

Returns one array of grid indices, `grid.shape = (len(dimensions),) + tuple(dimensions)`.

**If sparse is True:**

Returns a tuple of arrays, with `grid[i].shape = (1, ..., 1, dimensions[i], 1, ..., 1)` with dimensions[i] in the ith place

**See also:**

`mgrid`, `ogrid`, `meshgrid`

## Notes

The output shape in the dense case is obtained by prepending the number of dimensions in front of the tuple of dimensions, i.e. if *dimensions* is a tuple `(r0, ..., rN-1)` of length `N`, the output shape is `(N, r0, ..., rN-1)`.

The subarrays `grid[k]` contains the N-D array of indices along the `k-th` axis. Explicitly:

```
grid[k, i0, i1, ..., iN-1] = ik
```

## Examples

```
>>> grid = np.indices((2, 3))
>>> grid.shape
(2, 2, 3)
>>> grid[0]        # row indices
array([[0, 0, 0],
       [1, 1, 1]])
>>> grid[1]        # column indices
array([[0, 1, 2],
       [0, 1, 2]])
```

The indices can be used as an index into an array.

```
>>> x = np.arange(20).reshape(5, 4)
>>> row, col = np.indices((2, 3))
>>> x[row, col]
array([[0, 1, 2],
       [4, 5, 6]])
```

Note that it would be more straightforward in the above example to extract the required elements directly with `x[:2, :3]`.

If sparse is set to true, the grid will be returned in a sparse representation.

```
>>> i, j = np.indices((2, 3), sparse=True)
>>> i.shape
(2, 1)
>>> j.shape
```

(continues on next page)

(continued from previous page)

```
(1, 3)
>>> i        # row indices
array([[0],
       [1]])
>>> j        # column indices
array([[0, 1, 2]])
```

`numpy.ma.`**`where`**`(`*condition*, *x=<no value>*, *y=<no value>*`)`

Return a masked array with elements from *x* or *y*, depending on condition.

---

**Note:** When only *condition* is provided, this function is identical to `nonzero`. The rest of this documentation covers only the case where all three arguments are provided.

---

**Parameters**

**condition**

[array_like, bool] Where True, yield *x*, otherwise yield *y*.

**x, y**

[array_like, optional] Values from which to choose. *x*, *y* and *condition* need to be broadcastable to some shape.

**Returns**

**out**

[MaskedArray] An masked array with `masked` elements where the condition is masked, elements from *x* where *condition* is True, and elements from *y* elsewhere.

**See also:**

**`numpy.where`**

Equivalent function in the top-level NumPy module.

**`nonzero`**

The function that is called when x and y are omitted

## Examples

```
>>> x = np.ma.array(np.arange(9.).reshape(3, 3), mask=[[0, 1, 0],
...                                                    [1, 0, 1],
...                                                    [0, 1, 0]])
>>> x
masked_array(
  data=[[0.0, --, 2.0],
        [--, 4.0, --],
        [6.0, --, 8.0]],
  mask=[[False,  True, False],
        [ True, False,  True],
        [False,  True, False]],
```

(continues on next page)

(continued from previous page)

```
  fill_value=1e+20)
>>> np.ma.where(x > 5, x, -3.1416)
masked_array(
  data=[[-3.1416, --, -3.1416],
        [--, -3.1416, --],
        [6.0, --, 8.0]],
  mask=[[False,  True, False],
        [ True, False,  True],
        [False,  True, False]],
  fill_value=1e+20)
```

## 1.8 The Array Interface

---

**Note:** This page describes the numpy-specific API for accessing the contents of a numpy array from other C extensions. **PEP 3118** – `The Revised Buffer Protocol` introduces similar, standardized API to Python 2.6 and 3.0 for any extension module to use. Cython's buffer array support uses the **PEP 3118** API; see the Cython numpy tutorial. Cython provides a way to write code that supports the buffer protocol with Python versions older than 2.6 because it has a backward-compatible implementation utilizing the array interface described here.

---

**version**

3

The array interface (sometimes called array protocol) was created in 2005 as a means for array-like Python objects to reuse each other's data buffers intelligently whenever possible. The homogeneous N-dimensional array interface is a default mechanism for objects to share N-dimensional array memory and information. The interface consists of a Python-side and a C-side using two attributes. Objects wishing to be considered an N-dimensional array in application code should support at least one of these attributes. Objects wishing to support an N-dimensional array in application code should look for at least one of these attributes and use the information provided appropriately.

This interface describes homogeneous arrays in the sense that each item of the array has the same "type". This type can be very simple or it can be a quite arbitrary and complicated C-like structure.

There are two ways to use the interface: A Python side and a C-side. Both are separate attributes.

### 1.8.1 Python side

This approach to the interface consists of the object having an *`__array_interface__`* attribute.

**`__array_interface__`**

A dictionary of items (3 required and 5 optional). The optional keys in the dictionary have implied defaults if they are not provided.

The keys are:

**shape** (required)

Tuple whose elements are the array size in each dimension. Each entry is an integer (a Python int or long). Note that these integers could be larger than the platform "int" or "long" could hold (a Python int is a C long). It is up to the code using this attribute to handle this appropriately; either by raising an error when overflow is possible, or by using `Py_LONG_LONG` as the C type for the shapes.

**typestr** (required)

A string providing the basic type of the homogenous array The basic string format consists of 3 parts: a character describing the byteorder of the data (<: little-endian, >: big-endian, `|`: not-relevant), a character code giving the basic type of the array, and an integer providing the number of bytes the type uses.

The basic type character codes are:

| | |
|---|---|
| `t` | Bit field (following integer gives the number of bits in the bit field). |
| `b` | Boolean (integer type where all values are only True or False) |
| `i` | Integer |
| `u` | Unsigned integer |
| `f` | Floating point |
| `c` | Complex floating point |
| `m` | Timedelta |
| `M` | Datetime |
| `O` | Object (i.e. the memory contains a pointer to `PyObject`) |
| `S` | String (fixed-length sequence of char) |
| `U` | Unicode (fixed-length sequence of `Py_UNICODE`) |
| `V` | Other (void * – each item is a fixed-size chunk of memory) |

**descr** (optional)

A list of tuples providing a more detailed description of the memory layout for each item in the homogeneous array. Each tuple in the list has two or three elements. Normally, this attribute would be used when *typestr* is `V[0-9]+`, but this is not a requirement. The only requirement is that the number of bytes represented in the *typestr* key is the same as the total number of bytes represented here. The idea is to support descriptions of C-like structs that make up array elements. The elements of each tuple in the list are

1. A string providing a name associated with this portion of the datatype. This could also be a tuple of `('full name', 'basic_name')` where basic name would be a valid Python variable name representing the full name of the field.
2. Either a basic-type description string as in *typestr* or another list (for nested structured types)
3. An optional shape tuple providing how many times this part of the structure should be repeated. No repeats are assumed if this is not given. Very complicated structures can be described using this generic interface. Notice, however, that each element of the array is still of the same data-type. Some examples of using this interface are given below.

**Default**: `[('', typestr)]`

**data** (optional)

A 2-tuple whose first argument is an integer (a long integer if necessary) that points to the data-area storing the array contents. This pointer must point to the first element of data (in other words any offset is always ignored in this case). The second entry in the tuple is a read-only flag (true means the data area is read-only).

This attribute can also be an object exposing the `buffer interface` which will be used to share the data. If this key is not present (or returns None), then memory sharing will be done through the buffer interface of the object itself. In this case, the offset key can be used to indicate the start of the buffer. A reference to the object exposing the array interface must be stored by the new object if the memory area is to be secured.

**Default**: None

**strides** (optional)

Either None to indicate a C-style contiguous array or a Tuple of strides which provides the number of bytes needed to jump to the next array element in the corresponding dimension. Each entry must be an integer (a Python `int` or `long`). As with shape, the values may be larger than can be represented by a C "int" or "long"; the calling code should handle this appropriately, either by raising an error, or by using `Py_LONG_LONG` in C. The default is None which implies a C-style contiguous memory buffer. In this model, the last dimension of the array varies the fastest. For example, the default strides tuple for an object whose array entries are 8 bytes long and whose shape is (10,20,30) would be (4800, 240, 8)

**Default**: None (C-style contiguous)

**mask** (optional)

None or an object exposing the array interface. All elements of the mask array should be interpreted only as true or not true indicating which elements of this array are valid. The shape of this object should be *"broadcastable"* to the shape of the original array.

**Default**: None (All array values are valid)

**offset** (optional)

An integer offset into the array data region. This can only be used when data is None or returns a `buffer` object.

**Default**: 0.

**version** (required)

An integer showing the version of the interface (i.e. 3 for this version). Be careful not to use this to invalidate objects exposing future versions of the interface.

## 1.8.2 C-struct access

This approach to the array interface allows for faster access to an array using only one attribute lookup and a well-defined C-structure.

**`__array_struct__`**

A :c:type: *PyCObject* whose `voidptr` member contains a pointer to a filled *`PyArrayInterface`* structure. Memory for the structure is dynamically created and the `PyCObject` is also created with an appropriate destructor so the retriever of this attribute simply has to apply `Py_DECREF` to the object returned by this attribute when it is finished. Also, either the data needs to be copied out, or a reference to the object exposing this attribute must be held to ensure the data is not freed. Objects exposing the `__array_struct__` interface must also not reallocate their memory if other objects are referencing them.

The PyArrayInterface structure is defined in `numpy/ndarrayobject.h` as:

```
typedef struct {
  int two;              /* contains the integer 2 -- simple sanity check */
  int nd;               /* number of dimensions */
  char typekind;        /* kind in array --- character code of typestr */
  int itemsize;         /* size of each element */
  int flags;            /* flags indicating how the data should be interpreted */
                        /*   must set ARR_HAS_DESCR bit to validate descr */
  Py_intptr_t *shape;   /* A length-nd array of shape information */
  Py_intptr_t *strides; /* A length-nd array of stride information */
  void *data;           /* A pointer to the first element of the array */
  PyObject *descr;      /* NULL or data-description (same as descr key
                                of __array_interface__) -- must set ARR_HAS_DESCR
```

(continues on next page)

(continued from previous page)

```
                                flag or this will be ignored. */
} PyArrayInterface;
```

The flags member may consist of 5 bits showing how the data should be interpreted and one bit showing how the Interface should be interpreted. The data-bits are `CONTIGUOUS` (0x1), `FORTRAN` (0x2), `ALIGNED` (0x100), `NOTSWAPPED` (0x200), and `WRITEABLE` (0x400). A final flag `ARR_HAS_DESCR` (0x800) indicates whether or not this structure has the arrdescr field. The field should not be accessed unless this flag is present.

---

**New since June 16, 2006:**

In the past most implementations used the "desc" member of the `PyCObject` itself (do not confuse this with the "descr" member of the *`PyArrayInterface`* structure above — they are two separate things) to hold the pointer to the object exposing the interface. This is now an explicit part of the interface. Be sure to own a reference to the object when the `PyCObject` is created using `PyCObject_FromVoidPtrAndDesc`.

---

## 1.8.3 Type description examples

For clarity it is useful to provide some examples of the type description and corresponding *`__array_interface__`* 'descr' entries. Thanks to Scott Gilbert for these examples:

In every case, the 'descr' key is optional, but of course provides more information which may be important for various applications:

```
* Float data
    typestr == '>f4'
    descr == [('','>f4')]

* Complex double
    typestr == '>c8'
    descr == [('real','>f4'), ('imag','>f4')]

* RGB Pixel data
    typestr == '|V3'
    descr == [('r','|u1'), ('g','|u1'), ('b','|u1')]

* Mixed endian (weird but could happen).
    typestr == '|V8' (or '>u8')
    descr == [('big','>i4'), ('little','<i4')]

* Nested structure
    struct {
        int ival;
        struct {
            unsigned short sval;
            unsigned char bval;
            unsigned char cval;
        } sub;
    }
    typestr == '|V8' (or '<u8' if you want)
    descr == [('ival','<i4'), ('sub', [('sval','<u2'), ('bval','|u1'), ('cval','|u1')␣
↪]) ]

* Nested array
```

(continues on next page)

(continued from previous page)

```
    struct {
        int ival;
        double data[16*4];
    }
    typestr == '|V516'
    descr == [('ival','>i4'), ('data','>f8',(16,4))]

* Padded structure
    struct {
        int ival;
        double dval;
    }
    typestr == '|V16'
    descr == [('ival','>i4'),('','|V4'),('dval','>f8')]
```

It should be clear that any structured type could be described using this interface.

## 1.8.4 Differences with Array interface (Version 2)

The version 2 interface was very similar. The differences were largely aesthetic. In particular:

1. The PyArrayInterface structure had no descr member at the end (and therefore no flag ARR_HAS_DESCR)
2. The desc member of the PyCObject returned from __array_struct__ was not specified. Usually, it was the object exposing the array (so that a reference to it could be kept and destroyed when the C-object was destroyed). Now it must be a tuple whose first element is a string with "PyArrayInterface Version #" and whose second element is the object exposing the array.
3. The tuple returned from __array_interface__['data'] used to be a hex-string (now it is an integer or a long integer).
4. There was no __array_interface__ attribute instead all of the keys (except for version) in the __array_interface__ dictionary were their own attribute: Thus to obtain the Python-side information you had to access separately the attributes:
    - __array_data__
    - __array_shape__
    - __array_strides__
    - __array_typestr__
    - __array_descr__
    - __array_offset__
    - __array_mask__

## 1.9 Datetimes and Timedeltas

New in version 1.7.0.

Starting in NumPy 1.7, there are core array data types which natively support datetime functionality. The data type is called "datetime64", so named because "datetime" is already taken by the datetime library included in Python.

**Note:** The datetime API is *experimental* in 1.7.0, and may undergo changes in future versions of NumPy.

### 1.9.1 Basic Datetimes

The most basic way to create datetimes is from strings in ISO 8601 date or datetime format. The unit for internal storage is automatically selected from the form of the string, and can be either a *date unit* or a *time unit*. The date units are years ('Y'), months ('M'), weeks ('W'), and days ('D'), while the time units are hours ('h'), minutes ('m'), seconds ('s'), milliseconds ('ms'), and some additional SI-prefix seconds-based units. The datetime64 data type also accepts the string "NAT", in any combination of lowercase/uppercase letters, for a "Not A Time" value.

**Example**

A simple ISO date:

```
>>> np.datetime64('2005-02-25')
numpy.datetime64('2005-02-25')
```

Using months for the unit:

```
>>> np.datetime64('2005-02')
numpy.datetime64('2005-02')
```

Specifying just the month, but forcing a 'days' unit:

```
>>> np.datetime64('2005-02', 'D')
numpy.datetime64('2005-02-01')
```

From a date and time:

```
>>> np.datetime64('2005-02-25T03:30')
numpy.datetime64('2005-02-25T03:30')
```

NAT (not a time):

```
>>> np.datetime64('nat')
numpy.datetime64('NaT')
```

When creating an array of datetimes from a string, it is still possible to automatically select the unit from the inputs, by using the datetime type with generic units.

**Example**

```
>>> np.array(['2007-07-13', '2006-01-13', '2010-08-13'], dtype='datetime64')
array(['2007-07-13', '2006-01-13', '2010-08-13'], dtype='datetime64[D]')
```

```
>>> np.array(['2001-01-01T12:00', '2002-02-03T13:56:03.172'], dtype='datetime64')
array(['2001-01-01T12:00:00.000', '2002-02-03T13:56:03.172'],
      dtype='datetime64[ms]')
```

The datetime type works with many common NumPy functions, for example *`arange`* can be used to generate ranges of dates.

**Example**

All the dates for one month:

```
>>> np.arange('2005-02', '2005-03', dtype='datetime64[D]')
array(['2005-02-01', '2005-02-02', '2005-02-03', '2005-02-04',
       '2005-02-05', '2005-02-06', '2005-02-07', '2005-02-08',
       '2005-02-09', '2005-02-10', '2005-02-11', '2005-02-12',
       '2005-02-13', '2005-02-14', '2005-02-15', '2005-02-16',
       '2005-02-17', '2005-02-18', '2005-02-19', '2005-02-20',
       '2005-02-21', '2005-02-22', '2005-02-23', '2005-02-24',
       '2005-02-25', '2005-02-26', '2005-02-27', '2005-02-28'],
      dtype='datetime64[D]')
```

The datetime object represents a single moment in time. If two datetimes have different units, they may still be representing the same moment of time, and converting from a bigger unit like months to a smaller unit like days is considered a 'safe' cast because the moment of time is still being represented exactly.

**Example**

```
>>> np.datetime64('2005') == np.datetime64('2005-01-01')
True
```

```
>>> np.datetime64('2010-03-14T15') == np.datetime64('2010-03-14T15:00:00.00')
True
```

Deprecated since version 1.11.0: NumPy does not store timezone information. For backwards compatibility, datetime64 still parses timezone offsets, which it handles by converting to UTC. This behaviour is deprecated and will raise an error in the future.

### 1.9.2 Datetime and Timedelta Arithmetic

NumPy allows the subtraction of two Datetime values, an operation which produces a number with a time unit. Because NumPy doesn't have a physical quantities system in its core, the timedelta64 data type was created to complement datetime64. The arguments for timedelta64 are a number, to represent the number of units, and a date/time unit, such as (D)ay, (M)onth, (Y)ear, (h)ours, (m)inutes, or (s)econds. The timedelta64 data type also accepts the string "NAT" in place of the number for a "Not A Time" value.

**Example**

```
>>> np.timedelta64(1, 'D')
numpy.timedelta64(1,'D')
```

```
>>> np.timedelta64(4, 'h')
numpy.timedelta64(4,'h')
```

```
>>> np.timedelta64('nAt')
numpy.timedelta64('NaT')
```

---

Datetimes and Timedeltas work together to provide ways for simple datetime calculations.

---

**Example**

```
>>> np.datetime64('2009-01-01') - np.datetime64('2008-01-01')
numpy.timedelta64(366,'D')
```

```
>>> np.datetime64('2009') + np.timedelta64(20, 'D')
numpy.datetime64('2009-01-21')
```

```
>>> np.datetime64('2011-06-15T00:00') + np.timedelta64(12, 'h')
numpy.datetime64('2011-06-15T12:00')
```

```
>>> np.timedelta64(1,'W') / np.timedelta64(1,'D')
7.0
```

```
>>> np.timedelta64(1,'W') % np.timedelta64(10,'D')
numpy.timedelta64(7,'D')
```

```
>>> np.datetime64('nat') - np.datetime64('2009-01-01')
numpy.timedelta64('NaT','D')
```

```
>>> np.datetime64('2009-01-01') + np.timedelta64('nat')
numpy.datetime64('NaT')
```

---

There are two Timedelta units ('Y', years and 'M', months) which are treated specially, because how much time they represent changes depending on when they are used. While a timedelta day unit is equivalent to 24 hours, there is no way to convert a month unit into days, because different months have different numbers of days.

---

**Example**

```
>>> a = np.timedelta64(1, 'Y')
```

```
>>> np.timedelta64(a, 'M')
numpy.timedelta64(12,'M')
```

```
>>> np.timedelta64(a, 'D')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Cannot cast NumPy timedelta64 scalar from metadata [Y] to [D] according to␣
↪the rule 'same_kind'
```

### 1.9.3 Datetime Units

The Datetime and Timedelta data types support a large number of time units, as well as generic units which can be coerced into any of the other units based on input data.

Datetimes are always stored based on POSIX time (though having a TAI mode which allows for accounting of leap-seconds is proposed), with an epoch of 1970-01-01T00:00Z. This means the supported dates are always a symmetric interval around the epoch, called "time span" in the table below.

The length of the span is the range of a 64-bit integer times the length of the date or unit. For example, the time span for 'W' (week) is exactly 7 times longer than the time span for 'D' (day), and the time span for 'D' (day) is exactly 24 times longer than the time span for 'h' (hour).

Here are the date units:

| Code | Meaning | Time span (relative) | Time span (absolute) |
|---|---|---|---|
| Y | year | +/- 9.2e18 years | [9.2e18 BC, 9.2e18 AD] |
| M | month | +/- 7.6e17 years | [7.6e17 BC, 7.6e17 AD] |
| W | week | +/- 1.7e17 years | [1.7e17 BC, 1.7e17 AD] |
| D | day | +/- 2.5e16 years | [2.5e16 BC, 2.5e16 AD] |

And here are the time units:

| Code | Meaning | Time span (relative) | Time span (absolute) |
|---|---|---|---|
| h | hour | +/- 1.0e15 years | [1.0e15 BC, 1.0e15 AD] |
| m | minute | +/- 1.7e13 years | [1.7e13 BC, 1.7e13 AD] |
| s | second | +/- 2.9e11 years | [2.9e11 BC, 2.9e11 AD] |
| ms | millisecond | +/- 2.9e8 years | [ 2.9e8 BC, 2.9e8 AD] |
| us | microsecond | +/- 2.9e5 years | [290301 BC, 294241 AD] |
| ns | nanosecond | +/- 292 years | [ 1678 AD, 2262 AD] |
| ps | picosecond | +/- 106 days | [ 1969 AD, 1970 AD] |
| fs | femtosecond | +/- 2.6 hours | [ 1969 AD, 1970 AD] |
| as | attosecond | +/- 9.2 seconds | [ 1969 AD, 1970 AD] |

### 1.9.4 Business Day Functionality

To allow the datetime to be used in contexts where only certain days of the week are valid, NumPy includes a set of "busday" (business day) functions.

The default for busday functions is that the only valid days are Monday through Friday (the usual business days). The implementation is based on a "weekmask" containing 7 Boolean flags to indicate valid days; custom weekmasks are possible that specify other sets of valid days.

The "busday" functions can additionally check a list of "holiday" dates, specific dates that are not valid days.

The function *busday_offset* allows you to apply offsets specified in business days to datetimes with a unit of 'D' (day).

---

**Example**

```
>>> np.busday_offset('2011-06-23', 1)
numpy.datetime64('2011-06-24')
```

```
>>> np.busday_offset('2011-06-23', 2)
numpy.datetime64('2011-06-27')
```

---

When an input date falls on the weekend or a holiday, *`busday_offset`* first applies a rule to roll the date to a valid business day, then applies the offset. The default rule is 'raise', which simply raises an exception. The rules most typically used are 'forward' and 'backward'.

---

**Example**

```
>>> np.busday_offset('2011-06-25', 2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: Non-business day date in busday_offset
```

```
>>> np.busday_offset('2011-06-25', 0, roll='forward')
numpy.datetime64('2011-06-27')
```

```
>>> np.busday_offset('2011-06-25', 2, roll='forward')
numpy.datetime64('2011-06-29')
```

```
>>> np.busday_offset('2011-06-25', 0, roll='backward')
numpy.datetime64('2011-06-24')
```

```
>>> np.busday_offset('2011-06-25', 2, roll='backward')
numpy.datetime64('2011-06-28')
```

---

In some cases, an appropriate use of the roll and the offset is necessary to get a desired answer.

---

**Example**

The first business day on or after a date:

```
>>> np.busday_offset('2011-03-20', 0, roll='forward')
numpy.datetime64('2011-03-21')
>>> np.busday_offset('2011-03-22', 0, roll='forward')
numpy.datetime64('2011-03-22')
```

The first business day strictly after a date:

```
>>> np.busday_offset('2011-03-20', 1, roll='backward')
numpy.datetime64('2011-03-21')
>>> np.busday_offset('2011-03-22', 1, roll='backward')
numpy.datetime64('2011-03-23')
```

---

The function is also useful for computing some kinds of days like holidays. In Canada and the U.S., Mother's day is on the second Sunday in May, which can be computed with a custom weekmask.

---

**Example**

```
>>> np.busday_offset('2012-05', 1, roll='forward', weekmask='Sun')
numpy.datetime64('2012-05-13')
```

When performance is important for manipulating many business dates with one particular choice of weekmask and holidays, there is an object *busdaycalendar* which stores the data necessary in an optimized form.

### np.is_busday():

To test a datetime64 value to see if it is a valid day, use *is_busday*.

**Example**

```
>>> np.is_busday(np.datetime64('2011-07-15'))  # a Friday
True
>>> np.is_busday(np.datetime64('2011-07-16')) # a Saturday
False
>>> np.is_busday(np.datetime64('2011-07-16'), weekmask="Sat Sun")
True
>>> a = np.arange(np.datetime64('2011-07-11'), np.datetime64('2011-07-18'))
>>> np.is_busday(a)
array([ True,  True,  True,  True,  True, False, False])
```

### np.busday_count():

To find how many valid days there are in a specified range of datetime64 dates, use *busday_count*:

**Example**

```
>>> np.busday_count(np.datetime64('2011-07-11'), np.datetime64('2011-07-18'))
5
>>> np.busday_count(np.datetime64('2011-07-18'), np.datetime64('2011-07-11'))
-5
```

If you have an array of datetime64 day values, and you want a count of how many of them are valid dates, you can do this:

**Example**

```
>>> a = np.arange(np.datetime64('2011-07-11'), np.datetime64('2011-07-18'))
>>> np.count_nonzero(np.is_busday(a))
5
```

### Custom Weekmasks

Here are several examples of custom weekmask values. These examples specify the “busday” default of Monday through Friday being valid days.

Some examples:

```
# Positional sequences; positions are Monday through Sunday.
# Length of the sequence must be exactly 7.
weekmask = [1, 1, 1, 1, 1, 0, 0]
# list or other sequence; 0 == invalid day, 1 == valid day
weekmask = "1111100"
# string '0' == invalid day, '1' == valid day

# string abbreviations from this list: Mon Tue Wed Thu Fri Sat Sun
weekmask = "Mon Tue Wed Thu Fri"
# any amount of whitespace is allowed; abbreviations are case-sensitive.
weekmask = "MonTue Wed  Thu\tFri"
```

### CHAPTER

### TWO

# CONSTANTS

NumPy includes several constants:

`numpy.`**`Inf`**

IEEE 754 floating point representation of (positive) infinity.

Use *`inf`* because *`Inf`*, *`Infinity`*, *`PINF`* and *`infty`* are aliases for *`inf`*. For more details, see *`inf`*.

#### See Also

inf

`numpy.`**`Infinity`**

IEEE 754 floating point representation of (positive) infinity.

Use *`inf`* because *`Inf`*, *`Infinity`*, *`PINF`* and *`infty`* are aliases for *`inf`*. For more details, see *`inf`*.

#### See Also

inf

`numpy.`**`NAN`**

IEEE 754 floating point representation of Not a Number (NaN).

*`NaN`* and *`NAN`* are equivalent definitions of *`nan`*. Please use *`nan`* instead of *`NAN`*.

#### See Also

nan

`numpy.`**`NINF`**

IEEE 754 floating point representation of negative infinity.

### Returns

**y**

[float] A floating point representation of negative infinity.

### See Also

isinf : Shows which elements are positive or negative infinity

isposinf : Shows which elements are positive infinity

isneginf : Shows which elements are negative infinity

isnan : Shows which elements are Not a Number

isfinite : Shows which elements are finite (not one of Not a Number, positive infinity and negative infinity)

### Notes

NumPy uses the IEEE Standard for Binary Floating-Point for Arithmetic (IEEE 754). This means that Not a Number is not equivalent to infinity. Also that positive infinity is not equivalent to negative infinity. But infinity is equivalent to positive infinity.

### Examples

```
>>> np.NINF
-inf
>>> np.log(0)
-inf
```

`numpy.`**`NZERO`**

IEEE 754 floating point representation of negative zero.

### Returns

**y**

[float] A floating point representation of negative zero.

### See Also

PZERO : Defines positive zero.

isinf : Shows which elements are positive or negative infinity.

isposinf : Shows which elements are positive infinity.

isneginf : Shows which elements are negative infinity.

isnan : Shows which elements are Not a Number.

**isfinite**

[Shows which elements are finite - not one of] Not a Number, positive infinity and negative infinity.

### Notes

NumPy uses the IEEE Standard for Binary Floating-Point for Arithmetic (IEEE 754). Negative zero is considered to be a finite number.

### Examples

```
>>> np.NZERO
-0.0
>>> np.PZERO
0.0
```

```
>>> np.isfinite([np.NZERO])
array([ True])
>>> np.isnan([np.NZERO])
array([False])
>>> np.isinf([np.NZERO])
array([False])
```

`numpy.`**`NaN`**

IEEE 754 floating point representation of Not a Number (NaN).

`NaN` and `NAN` are equivalent definitions of `nan`. Please use `nan` instead of `NaN`.

### See Also

nan

`numpy.`**`PINF`**

IEEE 754 floating point representation of (positive) infinity.

Use `inf` because `Inf`, `Infinity`, `PINF` and `infty` are aliases for `inf`. For more details, see `inf`.

### See Also

inf

`numpy.`**`PZERO`**

IEEE 754 floating point representation of positive zero.

### Returns

**y**

[float] A floating point representation of positive zero.

### See Also

NZERO : Defines negative zero.

isinf : Shows which elements are positive or negative infinity.

isposinf : Shows which elements are positive infinity.

isneginf : Shows which elements are negative infinity.

isnan : Shows which elements are Not a Number.

**isfinite**

[Shows which elements are finite - not one of] Not a Number, positive infinity and negative infinity.

### Notes

NumPy uses the IEEE Standard for Binary Floating-Point for Arithmetic (IEEE 754). Positive zero is considered to be a finite number.

### Examples

```
>>> np.PZERO
0.0
>>> np.NZERO
-0.0
```

```
>>> np.isfinite([np.PZERO])
array([ True])
>>> np.isnan([np.PZERO])
array([False])
>>> np.isinf([np.PZERO])
array([False])
```

`numpy.`**`e`**

Euler's constant, base of natural logarithms, Napier's constant.

`e = 2.71828182845904523536028747135266249775724709369995...`

### See Also

exp : Exponential function log : Natural logarithm

### References

https://en.wikipedia.org/wiki/E_%28mathematical_constant%29

`numpy.`**`euler_gamma`**

`γ = 0.577215664901532860606512090082402431042...`

### References

https://en.wikipedia.org/wiki/Euler-Mascheroni_constant

`numpy.`**`inf`**

IEEE 754 floating point representation of (positive) infinity.

### Returns

**y**

[float] A floating point representation of positive infinity.

### See Also

isinf : Shows which elements are positive or negative infinity

isposinf : Shows which elements are positive infinity

isneginf : Shows which elements are negative infinity

isnan : Shows which elements are Not a Number

isfinite : Shows which elements are finite (not one of Not a Number, positive infinity and negative infinity)

### Notes

NumPy uses the IEEE Standard for Binary Floating-Point for Arithmetic (IEEE 754). This means that Not a Number is not equivalent to infinity. Also that positive infinity is not equivalent to negative infinity. But infinity is equivalent to positive infinity.

*`Inf`*, *`Infinity`*, *`PINF`* and *`infty`* are aliases for *`inf`*.

### Examples

```
>>> np.inf
inf
>>> np.array([1]) / 0.
array([ Inf])
```

`numpy.`**`infty`**

IEEE 754 floating point representation of (positive) infinity.

Use *`inf`* because *`Inf`*, *`Infinity`*, *`PINF`* and *`infty`* are aliases for *`inf`*. For more details, see *`inf`*.

### See Also

inf

`numpy.`**`nan`**

IEEE 754 floating point representation of Not a Number (NaN).

### Returns

y : A floating point representation of Not a Number.

### See Also

isnan : Shows which elements are Not a Number.

isfinite : Shows which elements are finite (not one of Not a Number, positive infinity and negative infinity)

### Notes

NumPy uses the IEEE Standard for Binary Floating-Point for Arithmetic (IEEE 754). This means that Not a Number is not equivalent to infinity.

*`NaN`* and *`NAN`* are aliases of *`nan`*.

### Examples

```
>>> np.nan
nan
>>> np.log(-1)
nan
>>> np.log([-1, 1, 2])
array([        NaN,  0.        ,  0.69314718])
```

`numpy.`**`newaxis`**

A convenient alias for None, useful for indexing arrays.

### See Also

`numpy.doc.indexing`

### Examples

```
>>> newaxis is None
True
>>> x = np.arange(3)
>>> x
array([0, 1, 2])
>>> x[:, newaxis]
array([[0],
[1],
```

(continues on next page)

(continued from previous page)

```
[2]])
>>> x[:, newaxis, newaxis]
array([[[0]],
[[1]],
[[2]]])
>>> x[:, newaxis] * x
array([[0, 0, 0],
[0, 1, 2],
[0, 2, 4]])
```

Outer product, same as `outer(x, y)`:

```
>>> y = np.arange(3, 6)
>>> x[:, newaxis] * y
array([[ 0,  0,  0],
[ 3,  4,  5],
[ 6,  8, 10]])
```

`x[newaxis, :]` is equivalent to `x[newaxis]` and `x[None]`:

```
>>> x[newaxis, :].shape
(1, 3)
>>> x[newaxis].shape
(1, 3)
>>> x[None].shape
(1, 3)
>>> x[:, newaxis].shape
(3, 1)
```

`numpy.`**`pi`**

`pi = 3.1415926535897932384626433...`

### References

https://en.wikipedia.org/wiki/Pi

## CHAPTER

## THREE

# UNIVERSAL FUNCTIONS (`UFUNC`)

A universal function (or ufunc for short) is a function that operates on `ndarrays` in an element-by-element fashion, supporting *array broadcasting*, *type casting*, and several other standard features. That is, a ufunc is a "vectorized" wrapper for a function that takes a fixed number of specific inputs and produces a fixed number of specific outputs.

In NumPy, universal functions are instances of the `numpy.ufunc` class. Many of the built-in functions are implemented in compiled C code. The basic ufuncs operate on scalars, but there is also a generalized kind for which the basic elements are sub-arrays (vectors, matrices, etc.), and broadcasting is done over other dimensions. One can also produce custom `ufunc` instances using the `frompyfunc` factory function.

## 3.1 Broadcasting

Each universal function takes array inputs and produces array outputs by performing the core function element-wise on the inputs (where an element is generally a scalar, but can be a vector or higher-order sub-array for generalized ufuncs). Standard broadcasting rules are applied so that inputs not sharing exactly the same shapes can still be usefully operated on. Broadcasting can be understood by four rules:

1. All input arrays with `ndim` smaller than the input array of largest `ndim`, have 1's prepended to their shapes.
2. The size in each dimension of the output shape is the maximum of all the input sizes in that dimension.
3. An input can be used in the calculation if its size in a particular dimension either matches the output size in that dimension, or has value exactly 1.
4. If an input has a dimension size of 1 in its shape, the first data entry in that dimension will be used for all calculations along that dimension. In other words, the stepping machinery of the ufunc will simply not step along that dimension (the *stride* will be 0 for that dimension).

Broadcasting is used throughout NumPy to decide how to handle disparately shaped arrays; for example, all arithmetic operations (`+`, `-`, `*`, ...) between `ndarrays` broadcast the arrays before operation.

A set of arrays is called "broadcastable" to the same shape if the above rules produce a valid result, *i.e.*, one of the following is true:

1. The arrays all have exactly the same shape.
2. The arrays all have the same number of dimensions and the length of each dimensions is either a common length or 1.
3. The arrays that have too few dimensions can have their shapes prepended with a dimension of length 1 to satisfy property 2.

---

**Example**

If `a.shape` is (5,1), `b.shape` is (1,6), `c.shape` is (6,) and `d.shape` is () so that *d* is a scalar, then *a*, *b*, *c*, and *d* are all broadcastable to dimension (5,6); and

- *a* acts like a (5,6) array where `a[:,0]` is broadcast to the other columns,
- *b* acts like a (5,6) array where `b[0,:]` is broadcast to the other rows,
- *c* acts like a (1,6) array and therefore like a (5,6) array where `c[:]` is broadcast to every row, and finally,
- *d* acts like a (5,6) array where the single value is repeated.

## 3.2 Output type determination

The output of the ufunc (and its methods) is not necessarily an `ndarray`, if all input arguments are not `ndarrays`. Indeed, if any input defines an `__array_ufunc__` method, control will be passed completely to that function, i.e., the ufunc is overridden.

If none of the inputs overrides the ufunc, then all output arrays will be passed to the `__array_prepare__` and `__array_wrap__` methods of the input (besides `ndarrays`, and scalars) that defines it **and** has the highest `__array_priority__` of any other input to the universal function. The default `__array_priority__` of the ndarray is 0.0, and the default `__array_priority__` of a subtype is 0.0. Matrices have `__array_priority__` equal to 10.0.

All ufuncs can also take output arguments. If necessary, output will be cast to the data-type(s) of the provided output array(s). If a class with an `__array__` method is used for the output, results will be written to the object returned by `__array__`. Then, if the class also has an `__array_prepare__` method, it is called so metadata may be determined based on the context of the ufunc (the context consisting of the ufunc itself, the arguments passed to the ufunc, and the ufunc domain.) The array object returned by `__array_prepare__` is passed to the ufunc for computation. Finally, if the class also has an `__array_wrap__` method, the returned `ndarray` result will be passed to that method just before passing control back to the caller.

## 3.3 Use of internal buffers

Internally, buffers are used for misaligned data, swapped data, and data that has to be converted from one data type to another. The size of internal buffers is settable on a per-thread basis. There can be up to $2(n_{\text{inputs}} + n_{\text{outputs}})$ buffers of the specified size created to handle the data from all the inputs and outputs of a ufunc. The default size of a buffer is 10,000 elements. Whenever buffer-based calculation would be needed, but all input arrays are smaller than the buffer size, those misbehaved or incorrectly-typed arrays will be copied before the calculation proceeds. Adjusting the size of the buffer may therefore alter the speed at which ufunc calculations of various sorts are completed. A simple interface for setting this variable is accessible using the function

| | |
|---|---|
| `setbufsize`(size) | Set the size of the buffer used in ufuncs. |

`numpy.`**`setbufsize`**(*size*)

Set the size of the buffer used in ufuncs.

**Parameters**

**size**

[int] Size of buffer.

## 3.4 Error handling

Universal functions can trip special floating-point status registers in your hardware (such as divide-by-zero). If available on your platform, these registers will be regularly checked during calculation. Error handling is controlled on a per-thread basis, and can be configured using the functions

| | |
|---|---|
| `seterr`([all, divide, over, under, invalid]) | Set how floating-point errors are handled. |
| `seterrcall`(func) | Set the floating-point error callback function or log object. |

`numpy.`**`seterr`**(*all=None*, *divide=None*, *over=None*, *under=None*, *invalid=None*)

Set how floating-point errors are handled.

Note that operations on integer scalar types (such as `int16`) are handled like floating point, and are affected by these settings.

**Parameters**

**all**

[{'ignore', 'warn', 'raise', 'call', 'print', 'log'}, optional] Set treatment for all types of floating-point errors at once:

- ignore: Take no action when the exception occurs.
- warn: Print a *RuntimeWarning* (via the Python `warnings` module).
- raise: Raise a *FloatingPointError*.
- call: Call a function specified using the `seterrcall` function.
- print: Print a warning directly to `stdout`.
- log: Record error in a Log object specified by `seterrcall`.

The default is not to change the current behavior.

**divide**

[{'ignore', 'warn', 'raise', 'call', 'print', 'log'}, optional] Treatment for division by zero.

**over**

[{'ignore', 'warn', 'raise', 'call', 'print', 'log'}, optional] Treatment for floating-point overflow.

**under**

[{'ignore', 'warn', 'raise', 'call', 'print', 'log'}, optional] Treatment for floating-point underflow.

**invalid**

[{'ignore', 'warn', 'raise', 'call', 'print', 'log'}, optional] Treatment for invalid floating-point operation.

**Returns**

**old_settings**

[dict] Dictionary containing the old settings.

**See also:**

***seterrcall***

Set a callback function for the 'call' mode.

*geterr*, *geterrcall*, *errstate*

## Notes

The floating-point exceptions are defined in the IEEE 754 standard [1]:

- Division by zero: infinite result obtained from finite numbers.
- Overflow: result too large to be expressed.
- Underflow: result so close to zero that some precision was lost.
- Invalid operation: result is not an expressible number, typically indicates that a NaN was produced.

## Examples

```
>>> old_settings = np.seterr(all='ignore')  #seterr to known value
>>> np.seterr(over='raise')
{'divide': 'ignore', 'over': 'ignore', 'under': 'ignore', 'invalid': 'ignore'}
>>> np.seterr(**old_settings)  # reset to default
{'divide': 'ignore', 'over': 'raise', 'under': 'ignore', 'invalid': 'ignore'}
```

```
>>> np.int16(32000) * np.int16(3)
30464
>>> old_settings = np.seterr(all='warn', over='raise')
>>> np.int16(32000) * np.int16(3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
FloatingPointError: overflow encountered in short_scalars
```

```
>>> from collections import OrderedDict
>>> old_settings = np.seterr(all='print')
>>> OrderedDict(np.geterr())
OrderedDict([('divide', 'print'), ('over', 'print'), ('under', 'print'), ('invalid
↪', 'print')])
>>> np.int16(32000) * np.int16(3)
30464
```

`numpy.`**`seterrcall`**`(`*func*`)`

Set the floating-point error callback function or log object.

There are two ways to capture floating-point error messages. The first is to set the error-handler to 'call', using *seterr*. Then, set the function to call using this function.

The second is to set the error-handler to 'log', using *seterr*. Floating-point errors then trigger a call to the 'write' method of the provided object.

**Parameters**

**func**

[callable f(err, flag) or object with write method] Function to call upon floating-point errors ('call'-mode) or object whose 'write' method is used to log such message ('log'-mode).

The call function takes two arguments. The first is a string describing the type of error (such as "divide by zero", "overflow", "underflow", or "invalid value"), and the second is the status flag. The flag is a byte, whose four least-significant bits indicate the type of error, one of "divide", "over", "under", "invalid":

```
[0 0 0 0 divide over under invalid]
```

In other words, `flags = divide + 2*over + 4*under + 8*invalid`.

If an object is provided, its write method should take one argument, a string.

**Returns**

**h**

[callable, log instance or None] The old error handler.

**See also:**

*`seterr`*, *`geterr`*, *`geterrcall`*

## Examples

Callback upon error:

```
>>> def err_handler(type, flag):
...     print("Floating point error (%s), with flag %s" % (type, flag))
...
```

```
>>> saved_handler = np.seterrcall(err_handler)
>>> save_err = np.seterr(all='call')
>>> from collections import OrderedDict
```

```
>>> np.array([1, 2, 3]) / 0.0
Floating point error (divide by zero), with flag 1
array([inf, inf, inf])
```

```
>>> np.seterrcall(saved_handler)
<function err_handler at 0x...>
>>> OrderedDict(sorted(np.seterr(**save_err).items()))
OrderedDict([('divide', 'call'), ('invalid', 'call'), ('over', 'call'), ('under',
↪'call')])
```

Log error message:

```
>>> class Log:
...     def write(self, msg):
...         print("LOG: %s" % msg)
...
```

```
>>> log = Log()
>>> saved_handler = np.seterrcall(log)
>>> save_err = np.seterr(all='log')
>>> np.array([1, 2, 3]) / 0.0
LOG: Warning: divide by zero encountered in true_divide
array([inf, inf, inf])
```

```
>>> np.seterrcall(saved_handler)
<numpy.core.numeric.Log object at 0x...>
>>> OrderedDict(sorted(np.seterr(**save_err).items()))
OrderedDict([('divide', 'log'), ('invalid', 'log'), ('over', 'log'), ('under',
↪'log')])
```

## 3.5 Casting Rules

**Note:** In NumPy 1.6.0, a type promotion API was created to encapsulate the mechanism for determining output types. See the functions `result_type`, `promote_types`, and `min_scalar_type` for more details.

At the core of every ufunc is a one-dimensional strided loop that implements the actual function for a specific type combination. When a ufunc is created, it is given a static list of inner loops and a corresponding list of type signatures over which the ufunc operates. The ufunc machinery uses this list to determine which inner loop to use for a particular case. You can inspect the `.types` attribute for a particular ufunc to see which type combinations have a defined inner loop and which output type they produce (*character codes* are used in said output for brevity).

Casting must be done on one or more of the inputs whenever the ufunc does not have a core loop implementation for the input types provided. If an implementation for the input types cannot be found, then the algorithm searches for an implementation with a type signature to which all of the inputs can be cast "safely." The first one it finds in its internal list of loops is selected and performed, after all necessary type casting. Recall that internal copies during ufuncs (even for casting) are limited to the size of an internal buffer (which is user settable).

**Note:** Universal functions in NumPy are flexible enough to have mixed type signatures. Thus, for example, a universal function could be defined that works with floating-point and integer values. See `ldexp` for an example.

By the above description, the casting rules are essentially implemented by the question of when a data type can be cast "safely" to another data type. The answer to this question can be determined in Python with a function call: `can_cast(fromtype, totype)`. The Figure below shows the results of this call for the 24 internally supported types on the author's 64-bit system. You can generate this table for your system with the code given in the Figure.

**Figure**

Code segment showing the "can cast safely" table for a 64-bit system. Generally the output depends on the system; your system might result in a different table.

```
>>> mark = {False: ' -', True: ' Y'}
>>> def print_table(ntypes):
...     print('X ' + ' '.join(ntypes))
...     for row in ntypes:
...         print(row, end='')
...         for col in ntypes:
...             print(mark[np.can_cast(row, col)], end='')
...         print()
...
>>> print_table(np.typecodes['All'])
```

(continues on next page)

(continued from previous page)

```
X ? b h i l q p B H I L Q P e f d g F D G S U V O M m
? Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y - Y
b - Y Y Y Y Y Y - - - - - - Y Y Y Y Y Y Y Y Y Y Y - Y
h - - Y Y Y Y Y - - - - - - - Y Y Y Y Y Y Y Y Y Y - Y
i - - - Y Y Y Y - - - - - - - - Y Y - Y Y Y Y Y Y - Y
l - - - - Y Y Y - - - - - - - - Y Y - Y Y Y Y Y Y - Y
q - - - - Y Y Y - - - - - - - - Y Y - Y Y Y Y Y Y - Y
p - - - - Y Y Y - - - - - - - - Y Y - Y Y Y Y Y Y - Y
B - - Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y - Y
H - - - Y Y Y Y - Y Y Y Y Y - Y Y Y Y Y Y Y Y Y Y - Y
I - - - - Y Y Y - - Y Y Y Y - - Y Y - Y Y Y Y Y Y - Y
L - - - - - - - - - - Y Y Y - - Y Y - Y Y Y Y Y Y - -
Q - - - - - - - - - - Y Y Y - - Y Y - Y Y Y Y Y Y - -
P - - - - - - - - - - Y Y Y - - Y Y - Y Y Y Y Y Y - -
e - - - - - - - - - - - - - Y Y Y Y Y Y Y Y Y Y Y - -
f - - - - - - - - - - - - - - Y Y Y Y Y Y Y Y Y Y - -
d - - - - - - - - - - - - - - - Y Y - Y Y Y Y Y Y - -
g - - - - - - - - - - - - - - - - Y - - Y Y Y Y Y - -
F - - - - - - - - - - - - - - - - - Y Y Y Y Y Y Y - -
D - - - - - - - - - - - - - - - - - - Y Y Y Y Y Y - -
G - - - - - - - - - - - - - - - - - - - Y Y Y Y Y - -
S - - - - - - - - - - - - - - - - - - - - Y Y Y Y - -
U - - - - - - - - - - - - - - - - - - - - - Y Y Y - -
V - - - - - - - - - - - - - - - - - - - - - - Y Y - -
O - - - - - - - - - - - - - - - - - - - - - - Y Y - -
M - - - - - - - - - - - - - - - - - - - - - - Y Y Y -
m - - - - - - - - - - - - - - - - - - - - - - Y Y - Y
```

---

You should note that, while included in the table for completeness, the 'S', 'U', and 'V' types cannot be operated on by ufuncs. Also, note that on a 32-bit system the integer types may have different sizes, resulting in a slightly altered table.

Mixed scalar-array operations use a different set of casting rules that ensure that a scalar cannot "upcast" an array unless the scalar is of a fundamentally different kind of data (*i.e.*, under a different hierarchy in the data-type hierarchy) than the array. This rule enables you to use scalar constants in your code (which, as Python types, are interpreted accordingly in ufuncs) without worrying about whether the precision of the scalar constant will cause upcasting on your large (small precision) array.

## 3.6 Overriding Ufunc behavior

Classes (including ndarray subclasses) can override how ufuncs act on them by defining certain special methods. For details, see *Standard array subclasses*.

## 3.7 `ufunc`

### 3.7.1 Optional keyword arguments

All ufuncs take optional keyword arguments. Most of these represent advanced usage and will not typically be used.

*out*

New in version 1.6.

The first output can be provided as either a positional or a keyword parameter. Keyword 'out' arguments are incompatible with positional ones.

New in version 1.10.

The 'out' keyword argument is expected to be a tuple with one entry per output (which can be None for arrays to be allocated by the ufunc). For ufuncs with a single output, passing a single array (instead of a tuple holding a single array) is also valid.

Passing a single array in the 'out' keyword argument to a ufunc with multiple outputs is deprecated, and will raise a warning in numpy 1.10, and an error in a future release.

If 'out' is None (the default), a uninitialized return array is created. The output array is then filled with the results of the ufunc in the places that the broadcast 'where' is True. If 'where' is the scalar True (the default), then this corresponds to the entire output being filled. Note that outputs not explicitly filled are left with their uninitialized values.

*where*

New in version 1.7.

Accepts a boolean array which is broadcast together with the operands. Values of True indicate to calculate the ufunc at that position, values of False indicate to leave the value in the output alone. This argument cannot be used for generalized ufuncs as those take non-scalar input.

Note that if an uninitialized return array is created, values of False will leave those values **uninitialized**.

*axes*

New in version 1.15.

A list of tuples with indices of axes a generalized ufunc should operate on. For instance, for a signature of `(i,j),(j,k)->(i,k)` appropriate for matrix multiplication, the base elements are two-dimensional matrices and these are taken to be stored in the two last axes of each argument. The corresponding axes keyword would be `[(-2, -1), (-2, -1), (-2, -1)]`. For simplicity, for generalized ufuncs that operate on 1-dimensional arrays (vectors), a single integer is accepted instead of a single-element tuple, and for generalized ufuncs for which all outputs are scalars, the output tuples can be omitted.

*axis*

New in version 1.15.

A single axis over which a generalized ufunc should operate. This is a short-cut for ufuncs that operate over a single, shared core dimension, equivalent to passing in `axes` with entries of `(axis,)` for each single-core-dimension argument and `()` for all others. For instance, for a signature `(i),(i)->()`, it is equivalent to passing in `axes=[(axis,), (axis,), ()]`.

*keepdims*

New in version 1.15.

If this is set to *True*, axes which are reduced over will be left in the result as a dimension with size one, so that the result will broadcast correctly against the inputs. This option can only be used for generalized ufuncs that operate on inputs that all have the same number of core dimensions and with outputs that have no core dimensions, i.e., with signatures like `(i),(i)->()` or `(m,m)->()`. If used, the location of the dimensions in the output can be controlled with `axes` and `axis`.

*casting*

New in version 1.6.

May be 'no', 'equiv', 'safe', 'same_kind', or 'unsafe'. See `can_cast` for explanations of the parameter values.

Provides a policy for what kind of casting is permitted. For compatibility with previous versions of NumPy, this defaults to 'unsafe' for numpy < 1.7. In numpy 1.7 a transition to 'same_kind' was begun where ufuncs produce a DeprecationWarning for calls which are allowed under the 'unsafe' rules, but not under the 'same_kind' rules. From numpy 1.10 and onwards, the default is 'same_kind'.

*order*

New in version 1.6.

Specifies the calculation iteration order/memory layout of the output array. Defaults to 'K'. 'C' means the output should be C-contiguous, 'F' means F-contiguous, 'A' means F-contiguous if the inputs are F-contiguous and not also not C-contiguous, C-contiguous otherwise, and 'K' means to match the element ordering of the inputs as closely as possible.

*dtype*

New in version 1.6.

Overrides the dtype of the calculation and output arrays. Similar to *signature*.

*subok*

New in version 1.6.

Defaults to true. If set to false, the output will always be a strict array, not a subtype.

*signature*

Either a data-type, a tuple of data-types, or a special signature string indicating the input and output types of a ufunc. This argument allows you to provide a specific signature for the 1-d loop to use in the underlying calculation. If the loop specified does not exist for the ufunc, then a TypeError is raised. Normally, a suitable loop is found automatically by comparing the input types with what is available and searching for a loop with data-types to which all inputs can be cast safely. This keyword argument lets you bypass that search and choose a particular loop. A list of available signatures is provided by the **types** attribute of the ufunc object. For backwards compatibility this argument can also be provided as *sig*, although the long form is preferred. Note that this should not be confused with the generalized ufunc *signature* that is stored in the **signature** attribute of the of the ufunc object.

*extobj*

a list of length 3 specifying the ufunc buffer-size, the error mode integer, and the error call-back function. Normally, these values are looked up in a thread-specific dictionary. Passing them here circumvents that look up and uses the low-level specification provided for the error mode. This may be useful, for example, as an optimization for calculations requiring many ufunc calls on small arrays in a loop.

## 3.7.2 Attributes

There are some informational attributes that universal functions possess. None of the attributes can be set.

| | |
|---|---|
| **__doc__** | A docstring for each ufunc. The first part of the docstring is dynamically generated from the number of outputs, the name, and the number of inputs. The second part of the docstring is provided at creation time and stored with the ufunc. |
| **__name__** | The name of the ufunc. |

| | |
|---|---|
| *ufunc.nin* | The number of inputs. |
| *ufunc.nout* | The number of outputs. |
| *ufunc.nargs* | The number of arguments. |
| *ufunc.ntypes* | The number of types. |
| *ufunc.types* | Returns a list with types grouped input->output. |
| *ufunc.identity* | The identity value. |
| *ufunc.signature* | Definition of the core elements a generalized ufunc operates on. |

attribute

ufunc.**nin**

The number of inputs.

Data attribute containing the number of arguments the ufunc treats as input.

### Examples

```
>>> np.add.nin
2
>>> np.multiply.nin
2
>>> np.power.nin
2
>>> np.exp.nin
1
```

attribute

ufunc.**nout**

The number of outputs.

Data attribute containing the number of arguments the ufunc treats as output.

### Notes

Since all ufuncs can take output arguments, this will always be (at least) 1.

### Examples

```
>>> np.add.nout
1
>>> np.multiply.nout
1
>>> np.power.nout
1
>>> np.exp.nout
1
```

attribute

`ufunc.`**`nargs`**

The number of arguments.

Data attribute containing the number of arguments the ufunc takes, including optional ones.

### Notes

Typically this value will be one more than what you might expect because all ufuncs take the optional "out" argument.

### Examples

```
>>> np.add.nargs
3
>>> np.multiply.nargs
3
>>> np.power.nargs
3
>>> np.exp.nargs
2
```

attribute

`ufunc.`**`ntypes`**

The number of types.

The number of numerical NumPy types - of which there are 18 total - on which the ufunc can operate.

**See also:**

*`numpy.ufunc.types`*

### Examples

```
>>> np.add.ntypes
18
>>> np.multiply.ntypes
18
>>> np.power.ntypes
17
>>> np.exp.ntypes
7
```

(continues on next page)

(continued from previous page)

```
>>> np.remainder.ntypes
14
```

attribute

**ufunc.types**

Returns a list with types grouped input->output.

Data attribute listing the data-type “Domain-Range” groupings the ufunc can deliver. The data-types are given using the character codes.

**See also:**

*numpy.ufunc.ntypes*

### Examples

```
>>> np.add.types
['??->?', 'bb->b', 'BB->B', 'hh->h', 'HH->H', 'ii->i', 'II->I', 'll->l',
'LL->L', 'qq->q', 'QQ->Q', 'ff->f', 'dd->d', 'gg->g', 'FF->F', 'DD->D',
'GG->G', 'OO->O']
```

```
>>> np.multiply.types
['??->?', 'bb->b', 'BB->B', 'hh->h', 'HH->H', 'ii->i', 'II->I', 'll->l',
'LL->L', 'qq->q', 'QQ->Q', 'ff->f', 'dd->d', 'gg->g', 'FF->F', 'DD->D',
'GG->G', 'OO->O']
```

```
>>> np.power.types
['bb->b', 'BB->B', 'hh->h', 'HH->H', 'ii->i', 'II->I', 'll->l', 'LL->L',
'qq->q', 'QQ->Q', 'ff->f', 'dd->d', 'gg->g', 'FF->F', 'DD->D', 'GG->G',
'OO->O']
```

```
>>> np.exp.types
['f->f', 'd->d', 'g->g', 'F->F', 'D->D', 'G->G', 'O->O']
```

```
>>> np.remainder.types
['bb->b', 'BB->B', 'hh->h', 'HH->H', 'ii->i', 'II->I', 'll->l', 'LL->L',
'qq->q', 'QQ->Q', 'ff->f', 'dd->d', 'gg->g', 'OO->O']
```

attribute

**ufunc.identity**

The identity value.

Data attribute containing the identity element for the ufunc, if it has one. If it does not, the attribute value is None.

### Examples

```
>>> np.add.identity
0
>>> np.multiply.identity
1
>>> np.power.identity
1
>>> print(np.exp.identity)
None
```

attribute

`ufunc.`**`signature`**

Definition of the core elements a generalized ufunc operates on.

The signature determines how the dimensions of each input/output array are split into core and loop dimensions:

1. Each dimension in the signature is matched to a dimension of the corresponding passed-in array, starting from the end of the shape tuple.
2. Core dimensions assigned to the same label in the signature must have exactly matching sizes, no broadcasting is performed.
3. The core dimensions are removed from all inputs and the remaining dimensions are broadcast together, defining the loop dimensions.

### Notes

Generalized ufuncs are used internally in many linalg functions, and in the testing suite; the examples below are taken from these. For ufuncs that operate on scalars, the signature is None, which is equivalent to ‘()’ for every argument.

### Examples

```
>>> np.core.umath_tests.matrix_multiply.signature
'(m,n),(n,p)->(m,p)'
>>> np.linalg._umath_linalg.det.signature
'(m,m)->()'
>>> np.add.signature is None
True  # equivalent to '(),()->()'
```

## 3.7.3 Methods

All ufuncs have four methods. However, these methods only make sense on scalar ufuncs that take two input arguments and return one output argument. Attempting to call these methods on other ufuncs will cause a `ValueError`. The reduce-like methods all take an *axis* keyword, a *dtype* keyword, and an *out* keyword, and the arrays must all have dimension >= 1. The *axis* keyword specifies the axis of the array over which the reduction will take place (with negative values counting backwards). Generally, it is an integer, though for *`ufunc.reduce`*, it can also be a tuple of `int` to reduce over several axes at once, or None, to reduce over all axes. The *dtype* keyword allows you to manage a very common problem that arises when naively using *`ufunc.reduce`*. Sometimes you may have an array of a certain data type and wish to add up all of its elements, but the result does not fit into the data type of the array. This commonly happens if you have an array of single-byte integers. The *dtype* keyword allows you to alter the data type over which the reduction takes place (and therefore the type of the output). Thus, you can ensure that the output is a data type with precision large enough to handle your output. The responsibility of altering the reduce type is mostly up to you. There is one exception: if no *dtype* is given for a reduction on the "add" or "multiply" operations, then if the input type is an integer (or Boolean) data-type and smaller than the size of the `int_` data type, it will be internally upcast to the `int_` (or `uint`) data-type. Finally, the *out* keyword allows you to provide an output array (for single-output ufuncs, which are currently the only ones supported; for future extension, however, a tuple with a single argument can be passed in). If *out* is given, the *dtype* argument is ignored.

Ufuncs also have a fifth method that allows in place operations to be performed using fancy indexing. No buffering is used on the dimensions where fancy indexing is used, so the fancy index can list an item more than once and the operation will be performed on the result of the previous operation for that item.

| | |
|---|---|
| `ufunc.reduce`(a[, axis, dtype, out, …]) | Reduces *a*'s dimension by one, by applying ufunc along one axis. |
| `ufunc.accumulate`(array[, axis, dtype, out]) | Accumulate the result of applying the operator to all elements. |
| `ufunc.reduceat`(a, indices[, axis, dtype, out]) | Performs a (local) reduce with specified slices over a single axis. |
| `ufunc.outer`(A, B, **kwargs) | Apply the ufunc *op* to all pairs (a, b) with a in *A* and b in *B*. |
| `ufunc.at`(a, indices[, b]) | Performs unbuffered in place operation on operand 'a' for elements specified by 'indices'. |

method

`ufunc.`**`reduce`**(*a*, *axis=0*, *dtype=None*, *out=None*, *keepdims=False*, *initial=<no value>*, *where=True*)

Reduces *a*'s dimension by one, by applying ufunc along one axis.

Let $a.shape = (N_0, ..., N_i, ..., N_{M-1})$. Then $ufunc.reduce(a, axis = i)[k_0, .., k_{i-1}, k_{i+1}, .., k_{M-1}]$ = the result of iterating *j* over $range(N_i)$, cumulatively applying ufunc to each $a[k_0, .., k_{i-1}, j, k_{i+1}, .., k_{M-1}]$. For a one-dimensional array, reduce produces results equivalent to:

```
r = op.identity # op = ufunc
for i in range(len(A)):
  r = op(r, A[i])
return r
```

For example, add.reduce() is equivalent to sum().

**Parameters**

**a**

[array_like] The array to act on.

**axis**

[None or int or tuple of ints, optional] Axis or axes along which a reduction is performed. The default (*axis* = 0) is perform a reduction over the first dimension of the input array. *axis* may be negative, in which case it counts from the last to the first axis.

New in version 1.7.0.

If this is None, a reduction is performed over all the axes. If this is a tuple of ints, a reduction is performed on multiple axes, instead of a single axis or all the axes as before.

For operations which are either not commutative or not associative, doing a reduction over multiple axes is not well-defined. The ufuncs do not currently raise an exception in this case, but will likely do so in the future.

**dtype**

[data-type code, optional] The type used to represent the intermediate results. Defaults to the data-type of the output array if this is provided, or the data-type of the input array if no output array is provided.

**out**

[ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If not provided or None, a freshly-allocated array is returned. For consistency with `ufunc.__call__`, if given as a keyword, this may be wrapped in a 1-element tuple.

Changed in version 1.13.0: Tuples are allowed for keyword argument.

**keepdims**

[bool, optional] If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the original *arr*.

New in version 1.7.0.

**initial**

[scalar, optional] The value with which to start the reduction. If the ufunc has no identity or the dtype is object, this defaults to None - otherwise it defaults to ufunc.identity. If `None` is given, the first element of the reduction is used, and an error is thrown if the reduction is empty.

New in version 1.15.0.

**where**

[array_like of bool, optional] A boolean array which is broadcasted to match the dimensions of *a*, and selects elements to include in the reduction. Note that for ufuncs like `minimum` that do not have an identity defined, one has to pass in also `initial`.

New in version 1.17.0.

**Returns**

**r**

[ndarray] The reduced array. If *out* was supplied, *r* is a reference to it.

## Examples

```
>>> np.multiply.reduce([2,3,5])
30
```

A multi-dimensional array example:

```
>>> X = np.arange(8).reshape((2,2,2))
>>> X
array([[[0, 1],
        [2, 3]],
       [[4, 5],
        [6, 7]]])
>>> np.add.reduce(X, 0)
array([[ 4,  6],
       [ 8, 10]])
>>> np.add.reduce(X) # confirm: default axis value is 0
```

(continues on next page)

(continued from previous page)

```
array([[ 4,  6],
       [ 8, 10]])
>>> np.add.reduce(X, 1)
array([[ 2,  4],
       [10, 12]])
>>> np.add.reduce(X, 2)
array([[ 1,  5],
       [ 9, 13]])
```

You can use the `initial` keyword argument to initialize the reduction with a different value, and `where` to select specific elements to include:

```
>>> np.add.reduce([10], initial=5)
15
>>> np.add.reduce(np.ones((2, 2, 2)), axis=(0, 2), initial=10)
array([14., 14.])
>>> a = np.array([10., np.nan, 10])
>>> np.add.reduce(a, where=~np.isnan(a))
20.0
```

Allows reductions of empty arrays where they would normally fail, i.e. for ufuncs without an identity.

```
>>> np.minimum.reduce([], initial=np.inf)
inf
>>> np.minimum.reduce([[1., 2.], [3., 4.]], initial=10., where=[True, False])
array([ 1., 10.])
>>> np.minimum.reduce([])
Traceback (most recent call last):
    ...
ValueError: zero-size array to reduction operation minimum which has no identity
```

method

`ufunc.`**`accumulate`**(*array*, *axis=0*, *dtype=None*, *out=None*)

Accumulate the result of applying the operator to all elements.

For a one-dimensional array, accumulate produces results equivalent to:

```
r = np.empty(len(A))
t = op.identity        # op = the ufunc being applied to A's  elements
for i in range(len(A)):
    t = op(t, A[i])
    r[i] = t
return r
```

For example, add.accumulate() is equivalent to np.cumsum().

For a multi-dimensional array, accumulate is applied along only one axis (axis zero by default; see Examples below) so repeated use is necessary if one wants to accumulate over multiple axes.

**Parameters**

**array**

[array_like] The array to act on.

**axis**

[int, optional] The axis along which to apply the accumulation; default is zero.

**dtype**

[data-type code, optional] The data-type used to represent the intermediate results. Defaults to the data-type of the output array if such is provided, or the the data-type of the input array if no output array is provided.

**out**

[ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If not provided or None, a freshly-allocated array is returned. For consistency with `ufunc.__call__`, if given as a keyword, this may be wrapped in a 1-element tuple.

Changed in version 1.13.0: Tuples are allowed for keyword argument.

**Returns**

**r**

[ndarray] The accumulated values. If *out* was supplied, *r* is a reference to *out*.

## Examples

1-D array examples:

```
>>> np.add.accumulate([2, 3, 5])
array([ 2,  5, 10])
>>> np.multiply.accumulate([2, 3, 5])
array([ 2,  6, 30])
```

2-D array examples:

```
>>> I = np.eye(2)
>>> I
array([[1.,  0.],
       [0.,  1.]])
```

Accumulate along axis 0 (rows), down columns:

```
>>> np.add.accumulate(I, 0)
array([[1.,  0.],
       [1.,  1.]])
>>> np.add.accumulate(I) # no axis specified = axis zero
array([[1.,  0.],
       [1.,  1.]])
```

Accumulate along axis 1 (columns), through rows:

```
>>> np.add.accumulate(I, 1)
array([[1.,  1.],
       [0.,  1.]])
```

method

`ufunc.`**`reduceat`**(*a*, *indices*, *axis=0*, *dtype=None*, *out=None*)

Performs a (local) reduce with specified slices over a single axis.

For i in `range(len(indices))`, *`reduceat`* computes `ufunc.reduce(a[indices[i]:indices[i+1]])`, which becomes the i-th generalized "row" parallel to *axis* in the final result (i.e., in a 2-D array, for example, if *axis = 0*, it becomes the i-th row, but if *axis = 1*, it becomes the i-th column). There are three exceptions to this:

- when `i = len(indices) - 1` (so for the last index), `indices[i+1] = a.shape[axis]`.
- if `indices[i] >= indices[i + 1]`, the i-th generalized "row" is simply `a[indices[i]]`.
- if `indices[i] >= len(a)` or `indices[i] < 0`, an error is raised.

The shape of the output depends on the size of *`indices`*, and may be larger than *a* (this happens if `len(indices) > a.shape[axis]`).

**Parameters**

**a**

[array_like] The array to act on.

**indices**

[array_like] Paired indices, comma separated (not colon), specifying slices to reduce.

**axis**

[int, optional] The axis along which to apply the reduceat.

**dtype**

[data-type code, optional] The type used to represent the intermediate results. Defaults to the data type of the output array if this is provided, or the data type of the input array if no output array is provided.

**out**

[ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If not provided or None, a freshly-allocated array is returned. For consistency with `ufunc.__call__`, if given as a keyword, this may be wrapped in a 1-element tuple.

Changed in version 1.13.0: Tuples are allowed for keyword argument.

**Returns**

**r**

[ndarray] The reduced values. If *out* was supplied, *r* is a reference to *out*.

## Notes

A descriptive example:

If *a* is 1-D, the function *ufunc.accumulate(a)* is the same as `ufunc.reduceat(a, indices)[::2]` where *`indices`* is `range(len(array) - 1)` with a zero placed in every other element: `indices = zeros(2 * len(a) - 1)`, `indices[1::2] = range(1, len(a))`.

Don't be fooled by this attribute's name: *reduceat(a)* is not necessarily smaller than *a*.

### Examples

To take the running sum of four successive values:

```
>>> np.add.reduceat(np.arange(8),[0,4, 1,5, 2,6, 3,7])[::2]
array([ 6, 10, 14, 18])
```

A 2-D example:

```
>>> x = np.linspace(0, 15, 16).reshape(4,4)
>>> x
array([[ 0.,   1.,   2.,   3.],
       [ 4.,   5.,   6.,   7.],
       [ 8.,   9.,  10.,  11.],
       [12.,  13.,  14.,  15.]])
```

```
# reduce such that the result has the following five rows:
# [row1 + row2 + row3]
# [row4]
# [row2]
# [row3]
# [row1 + row2 + row3 + row4]
```

```
>>> np.add.reduceat(x, [0, 3, 1, 2, 0])
array([[12.,  15.,  18.,  21.],
       [12.,  13.,  14.,  15.],
       [ 4.,   5.,   6.,   7.],
       [ 8.,   9.,  10.,  11.],
       [24.,  28.,  32.,  36.]])
```

```
# reduce such that result has the following two columns:
# [col1 * col2 * col3, col4]
```

```
>>> np.multiply.reduceat(x, [0, 3], 1)
array([[   0.,     3.],
       [ 120.,     7.],
       [ 720.,    11.],
       [2184.,    15.]])
```

method

`ufunc.`**`outer`**(*A*, *B*, ***kwargs*)

Apply the ufunc *op* to all pairs (a, b) with a in *A* and b in *B*.

Let `M = A.ndim`, `N = B.ndim`. Then the result, *C*, of `op.outer(A, B)` is an array of dimension M + N such that:

$$C[i_0, ..., i_{M-1}, j_0, ..., j_{N-1}] = op(A[i_0, ..., i_{M-1}], B[j_0, ..., j_{N-1}])$$

For *A* and *B* one-dimensional, this is equivalent to:

```
r = empty(len(A),len(B))
for i in range(len(A)):
    for j in range(len(B)):
        r[i,j] = op(A[i], B[j]) # op = ufunc in question
```

**Parameters**

**A**

[array_like] First array

**B**

[array_like] Second array

**kwargs**

[any] Arguments to pass on to the ufunc. Typically *dtype* or *out*.

**Returns**

**r**

[ndarray] Output array

**See also:**

***numpy.outer***

A less powerful version of `np.multiply.outer` that *ravel*s all inputs to 1D. This exists primarily for compatibility with old code.

***tensordot***

`np.tensordot(a, b, axes=((), ()))` and `np.multiply.outer(a, b)` behave same for all dimensions of a and b.

## Examples

```
>>> np.multiply.outer([1, 2, 3], [4, 5, 6])
array([[ 4,  5,  6],
       [ 8, 10, 12],
       [12, 15, 18]])
```

A multi-dimensional example:

```
>>> A = np.array([[1, 2, 3], [4, 5, 6]])
>>> A.shape
(2, 3)
>>> B = np.array([[1, 2, 3, 4]])
>>> B.shape
(1, 4)
>>> C = np.multiply.outer(A, B)
>>> C.shape; C
(2, 3, 1, 4)
array([[[[ 1,  2,  3,  4]],
        [[ 2,  4,  6,  8]],
        [[ 3,  6,  9, 12]]],
       [[[ 4,  8, 12, 16]],
        [[ 5, 10, 15, 20]],
        [[ 6, 12, 18, 24]]]])
```

method

`ufunc.`**`at`**(*a*, *indices*, *b=None*)

Performs unbuffered in place operation on operand 'a' for elements specified by 'indices'. For addition ufunc, this method is equivalent to `a[indices] += b`, except that results are accumulated for elements that are indexed more than once. For example, `a[[0,0]] += 1` will only increment the first element once because of buffering, whereas `add.at(a, [0,0], 1)` will increment the first element twice.

New in version 1.8.0.

**Parameters**

**a**

[array_like] The array to perform in place operation on.

**indices**

[array_like or tuple] Array like index object or slice object for indexing into first operand. If first operand has multiple dimensions, indices can be a tuple of array like index objects or slice objects.

**b**

[array_like] Second operand for ufuncs requiring two operands. Operand must be broadcastable over first operand after indexing or slicing.

## Examples

Set items 0 and 1 to their negative values:

```
>>> a = np.array([1, 2, 3, 4])
>>> np.negative.at(a, [0, 1])
>>> a
array([-1, -2,  3,  4])
```

Increment items 0 and 1, and increment item 2 twice:

```
>>> a = np.array([1, 2, 3, 4])
>>> np.add.at(a, [0, 1, 2, 2], 1)
>>> a
array([2, 3, 5, 4])
```

Add items 0 and 1 in first array to second array, and store results in first array:

```
>>> a = np.array([1, 2, 3, 4])
>>> b = np.array([1, 2])
>>> np.add.at(a, [0, 1], b)
>>> a
array([2, 4, 3, 4])
```

**Warning:** A reduce-like operation on an array with a data-type that has a range "too small" to handle the result will silently wrap. One should use *`dtype`* to increase the size of the data-type over which reduction takes place.

## 3.8 Available ufuncs

There are currently more than 60 universal functions defined in `numpy` on one or more types, covering a wide variety of operations. Some of these ufuncs are called automatically on arrays when the relevant infix notation is used (*e.g.*, `add(a, b)` is called internally when `a + b` is written and *a* or *b* is an `ndarray`). Nevertheless, you may still want to use the ufunc call in order to use the optional output argument(s) to place the output(s) in an object (or objects) of your choice.

Recall that each ufunc operates element-by-element. Therefore, each scalar ufunc will be described as if acting on a set of scalar inputs to return a set of scalar outputs.

**Note:** The ufunc still returns its output(s) even if you use the optional output argument(s).

### 3.8.1 Math operations

| | |
|---|---|
| `add`(x1, x2, /[, out, where, casting, order, …]) | Add arguments element-wise. |
| `subtract`(x1, x2, /[, out, where, casting, …]) | Subtract arguments, element-wise. |
| `multiply`(x1, x2, /[, out, where, casting, …]) | Multiply arguments element-wise. |
| `matmul`(x1, x2, /[, out, casting, order, …]) | Matrix product of two arrays. |
| `divide`(x1, x2, /[, out, where, casting, …]) | Returns a true division of the inputs, element-wise. |
| `logaddexp`(x1, x2, /[, out, where, casting, …]) | Logarithm of the sum of exponentiations of the inputs. |
| `logaddexp2`(x1, x2, /[, out, where, casting, …]) | Logarithm of the sum of exponentiations of the inputs in base-2. |
| `true_divide`(x1, x2, /[, out, where, …]) | Returns a true division of the inputs, element-wise. |
| `floor_divide`(x1, x2, /[, out, where, …]) | Return the largest integer smaller or equal to the division of the inputs. |
| `negative`(x, /[, out, where, casting, order, …]) | Numerical negative, element-wise. |
| `positive`(x, /[, out, where, casting, order, …]) | Numerical positive, element-wise. |
| `power`(x1, x2, /[, out, where, casting, …]) | First array elements raised to powers from second array, element-wise. |
| `float_power`(x1, x2, /[, out, where, …]) | First array elements raised to powers from second array, element-wise. |
| `remainder`(x1, x2, /[, out, where, casting, …]) | Return element-wise remainder of division. |
| `mod`(x1, x2, /[, out, where, casting, order, …]) | Return element-wise remainder of division. |
| `fmod`(x1, x2, /[, out, where, casting, …]) | Return the element-wise remainder of division. |
| `divmod`(x1, x2[, out1, out2], / [[, out, …]) | Return element-wise quotient and remainder simultaneously. |
| `absolute`(x, /[, out, where, casting, order, …]) | Calculate the absolute value element-wise. |
| `fabs`(x, /[, out, where, casting, order, …]) | Compute the absolute values element-wise. |
| `rint`(x, /[, out, where, casting, order, …]) | Round elements of the array to the nearest integer. |
| `sign`(x, /[, out, where, casting, order, …]) | Returns an element-wise indication of the sign of a number. |
| `heaviside`(x1, x2, /[, out, where, casting, …]) | Compute the Heaviside step function. |
| `conj`(x, /[, out, where, casting, order, …]) | Return the complex conjugate, element-wise. |
| `conjugate`(x, /[, out, where, casting, …]) | Return the complex conjugate, element-wise. |
| `exp`(x, /[, out, where, casting, order, …]) | Calculate the exponential of all elements in the input array. |
| `exp2`(x, /[, out, where, casting, order, …]) | Calculate *2**p* for all *p* in the input array. |
| `log`(x, /[, out, where, casting, order, …]) | Natural logarithm, element-wise. |
| `log2`(x, /[, out, where, casting, order, …]) | Base-2 logarithm of *x*. |

Continued on next page

Table 5 – continued from previous page

| | |
|---|---|
| `log10`(x, /[, out, where, casting, order, …]) | Return the base 10 logarithm of the input array, element-wise. |
| `expm1`(x, /[, out, where, casting, order, …]) | Calculate `exp(x) - 1` for all elements in the array. |
| `log1p`(x, /[, out, where, casting, order, …]) | Return the natural logarithm of one plus the input array, element-wise. |
| `sqrt`(x, /[, out, where, casting, order, …]) | Return the non-negative square-root of an array, element-wise. |
| `square`(x, /[, out, where, casting, order, …]) | Return the element-wise square of the input. |
| `cbrt`(x, /[, out, where, casting, order, …]) | Return the cube-root of an array, element-wise. |
| `reciprocal`(x, /[, out, where, casting, …]) | Return the reciprocal of the argument, element-wise. |
| `gcd`(x1, x2, /[, out, where, casting, order, …]) | Returns the greatest common divisor of `|x1|` and `|x2|` |
| `lcm`(x1, x2, /[, out, where, casting, order, …]) | Returns the lowest common multiple of `|x1|` and `|x2|` |

**Tip:** The optional output arguments can be used to help you save memory for large calculations. If your arrays are large, complicated expressions can take longer than absolutely necessary due to the creation and (later) destruction of temporary calculation spaces. For example, the expression `G = a * b + c` is equivalent to `t1 = A * B; G = T1 + C; del t1`. It will be more quickly executed as `G = A * B; add(G, C, G)` which is the same as `G = A * B; G += C`.

## 3.8.2 Trigonometric functions

All trigonometric functions use radians when an angle is called for. The ratio of degrees to radians is $180^{\circ}/\pi$.

| | |
|---|---|
| `sin`(x, /[, out, where, casting, order, …]) | Trigonometric sine, element-wise. |
| `cos`(x, /[, out, where, casting, order, …]) | Cosine element-wise. |
| `tan`(x, /[, out, where, casting, order, …]) | Compute tangent element-wise. |
| `arcsin`(x, /[, out, where, casting, order, …]) | Inverse sine, element-wise. |
| `arccos`(x, /[, out, where, casting, order, …]) | Trigonometric inverse cosine, element-wise. |
| `arctan`(x, /[, out, where, casting, order, …]) | Trigonometric inverse tangent, element-wise. |
| `arctan2`(x1, x2, /[, out, where, casting, …]) | Element-wise arc tangent of `x1/x2` choosing the quadrant correctly. |
| `hypot`(x1, x2, /[, out, where, casting, …]) | Given the "legs" of a right triangle, return its hypotenuse. |
| `sinh`(x, /[, out, where, casting, order, …]) | Hyperbolic sine, element-wise. |
| `cosh`(x, /[, out, where, casting, order, …]) | Hyperbolic cosine, element-wise. |
| `tanh`(x, /[, out, where, casting, order, …]) | Compute hyperbolic tangent element-wise. |
| `arcsinh`(x, /[, out, where, casting, order, …]) | Inverse hyperbolic sine element-wise. |
| `arccosh`(x, /[, out, where, casting, order, …]) | Inverse hyperbolic cosine, element-wise. |
| `arctanh`(x, /[, out, where, casting, order, …]) | Inverse hyperbolic tangent element-wise. |
| `degrees`(x, /[, out, where, casting, order, …]) | Convert angles from radians to degrees. |
| `radians`(x, /[, out, where, casting, order, …]) | Convert angles from degrees to radians. |
| `deg2rad`(x, /[, out, where, casting, order, …]) | Convert angles from degrees to radians. |
| `rad2deg`(x, /[, out, where, casting, order, …]) | Convert angles from radians to degrees. |

### 3.8.3 Bit-twiddling functions

These function all require integer arguments and they manipulate the bit-pattern of those arguments.

| | |
|---|---|
| `bitwise_and`(x1, x2, /[, out, where, …]) | Compute the bit-wise AND of two arrays element-wise. |
| `bitwise_or`(x1, x2, /[, out, where, casting, …]) | Compute the bit-wise OR of two arrays element-wise. |
| `bitwise_xor`(x1, x2, /[, out, where, …]) | Compute the bit-wise XOR of two arrays element-wise. |
| `invert`(x, /[, out, where, casting, order, …]) | Compute bit-wise inversion, or bit-wise NOT, element-wise. |
| `left_shift`(x1, x2, /[, out, where, casting, …]) | Shift the bits of an integer to the left. |
| `right_shift`(x1, x2, /[, out, where, …]) | Shift the bits of an integer to the right. |

### 3.8.4 Comparison functions

| | |
|---|---|
| `greater`(x1, x2, /[, out, where, casting, …]) | Return the truth value of (x1 > x2) element-wise. |
| `greater_equal`(x1, x2, /[, out, where, …]) | Return the truth value of (x1 >= x2) element-wise. |
| `less`(x1, x2, /[, out, where, casting, …]) | Return the truth value of (x1 < x2) element-wise. |
| `less_equal`(x1, x2, /[, out, where, casting, …]) | Return the truth value of (x1 =< x2) element-wise. |
| `not_equal`(x1, x2, /[, out, where, casting, …]) | Return (x1 != x2) element-wise. |
| `equal`(x1, x2, /[, out, where, casting, …]) | Return (x1 == x2) element-wise. |

**Warning:** Do not use the Python keywords `and` and `or` to combine logical array expressions. These keywords will test the truth value of the entire array (not element-by-element as you might expect). Use the bitwise operators & and | instead.

| | |
|---|---|
| `logical_and`(x1, x2, /[, out, where, …]) | Compute the truth value of x1 AND x2 element-wise. |
| `logical_or`(x1, x2, /[, out, where, casting, …]) | Compute the truth value of x1 OR x2 element-wise. |
| `logical_xor`(x1, x2, /[, out, where, …]) | Compute the truth value of x1 XOR x2, element-wise. |
| `logical_not`(x, /[, out, where, casting, …]) | Compute the truth value of NOT x element-wise. |

**Warning:** The bit-wise operators & and | are the proper way to perform element-by-element array comparisons. Be sure you understand the operator precedence: `(a > 2) & (a < 5)` is the proper syntax because `a > 2 & a < 5` will result in an error due to the fact that `2 & a` is evaluated first.

| | |
|---|---|
| `maximum`(x1, x2, /[, out, where, casting, …]) | Element-wise maximum of array elements. |

**Tip:** The Python function `max()` will find the maximum over a one-dimensional array, but it will do so using a slower sequence interface. The reduce method of the maximum ufunc is much faster. Also, the `max()` method will not give answers you might expect for arrays with greater than one dimension. The reduce method of minimum also allows you to compute a total minimum over an array.

| | |
|---|---|
| `minimum`(x1, x2, /[, out, where, casting, …]) | Element-wise minimum of array elements. |

**Warning:** the behavior of `maximum(a, b)` is different than that of `max(a, b)`. As a ufunc, `maximum(a, b)` performs an element-by-element comparison of *a* and *b* and chooses each element of the result according to which element in the two arrays is larger. In contrast, `max(a, b)` treats the objects *a* and *b* as a whole, looks at the (total) truth value of `a > b` and uses it to return either *a* or *b* (as a whole). A similar difference exists between `minimum(a, b)` and `min(a, b)`.

| | |
|---|---|
| `fmax`(x1, x2, /[, out, where, casting, …]) | Element-wise maximum of array elements. |
| `fmin`(x1, x2, /[, out, where, casting, …]) | Element-wise minimum of array elements. |

### 3.8.5 Floating functions

Recall that all of these functions work element-by-element over an array, returning an array output. The description details only a single operation.

| | |
|---|---|
| `isfinite`(x, /[, out, where, casting, order, …]) | Test element-wise for finiteness (not infinity or not Not a Number). |
| `isinf`(x, /[, out, where, casting, order, …]) | Test element-wise for positive or negative infinity. |
| `isnan`(x, /[, out, where, casting, order, …]) | Test element-wise for NaN and return result as a boolean array. |
| `isnat`(x, /[, out, where, casting, order, …]) | Test element-wise for NaT (not a time) and return result as a boolean array. |
| `fabs`(x, /[, out, where, casting, order, …]) | Compute the absolute values element-wise. |
| `signbit`(x, /[, out, where, casting, order, …]) | Returns element-wise True where signbit is set (less than zero). |
| `copysign`(x1, x2, /[, out, where, casting, …]) | Change the sign of x1 to that of x2, element-wise. |
| `nextafter`(x1, x2, /[, out, where, casting, …]) | Return the next floating-point value after x1 towards x2, element-wise. |
| `spacing`(x, /[, out, where, casting, order, …]) | Return the distance between x and the nearest adjacent number. |
| `modf`(x[, out1, out2], / [[, out, where, …]) | Return the fractional and integral parts of an array, element-wise. |
| `ldexp`(x1, x2, /[, out, where, casting, …]) | Returns x1 * 2**x2, element-wise. |
| `frexp`(x[, out1, out2], / [[, out, where, …]) | Decompose the elements of x into mantissa and twos exponent. |
| `fmod`(x1, x2, /[, out, where, casting, …]) | Return the element-wise remainder of division. |
| `floor`(x, /[, out, where, casting, order, …]) | Return the floor of the input, element-wise. |
| `ceil`(x, /[, out, where, casting, order, …]) | Return the ceiling of the input, element-wise. |
| `trunc`(x, /[, out, where, casting, order, …]) | Return the truncated value of the input, element-wise. |

## CHAPTER

## FOUR

# ROUTINES

In this chapter routine docstrings are presented, grouped by functionality. Many docstrings contain example code, which demonstrates basic usage of the routine. The examples assume that NumPy is imported with:

```
>>> import numpy as np
```

A convenient way to execute examples is the `%doctest_mode` mode of IPython, which allows for pasting of multi-line examples and preserves indentation.

## 4.1 Array creation routines

**See also:**

Array creation

### 4.1.1 Ones and zeros

| | |
|---|---|
| `empty`(shape[, dtype, order]) | Return a new array of given shape and type, without initializing entries. |
| `empty_like`(prototype[, dtype, order, subok, …]) | Return a new array with the same shape and type as a given array. |
| `eye`(N[, M, k, dtype, order]) | Return a 2-D array with ones on the diagonal and zeros elsewhere. |
| `identity`(n[, dtype]) | Return the identity array. |
| `ones`(shape[, dtype, order]) | Return a new array of given shape and type, filled with ones. |
| `ones_like`(a[, dtype, order, subok, shape]) | Return an array of ones with the same shape and type as a given array. |
| `zeros`(shape[, dtype, order]) | Return a new array of given shape and type, filled with zeros. |
| `zeros_like`(a[, dtype, order, subok, shape]) | Return an array of zeros with the same shape and type as a given array. |
| `full`(shape, fill_value[, dtype, order]) | Return a new array of given shape and type, filled with *fill_value*. |
| `full_like`(a, fill_value[, dtype, order, …]) | Return a full array with the same shape and type as a given array. |

`numpy.`**`empty`**(*shape*, *dtype=float*, *order='C'*)

Return a new array of given shape and type, without initializing entries.

**Parameters**

**shape**

[int or tuple of int] Shape of the empty array, e.g., `(2, 3)` or `2`.

**dtype**

[data-type, optional] Desired output data-type for the array, e.g, `numpy.int8`. Default is `numpy.float64`.

**order**

[{'C', 'F'}, optional, default: 'C'] Whether to store multi-dimensional data in row-major (C-style) or column-major (Fortran-style) order in memory.

**Returns**

**out**

[ndarray] Array of uninitialized (arbitrary) data of the given shape, dtype, and order. Object arrays will be initialized to None.

**See also:**

***empty_like***

Return an empty array with shape and type of input.

***ones***

Return a new array setting values to one.

***zeros***

Return a new array setting values to zero.

***full***

Return a new array of given shape filled with value.

## Notes

`empty`, unlike `zeros`, does not set the array values to zero, and may therefore be marginally faster. On the other hand, it requires the user to manually set all the values in the array, and should be used with caution.

## Examples

```
>>> np.empty([2, 2])
array([[ -9.74499359e+001,   6.69583040e-309],
       [  2.13182611e-314,   3.06959433e-309]])         #uninitialized
```

```
>>> np.empty([2, 2], dtype=int)
array([[-1073741821, -1067949133],
       [  496041986,    19249760]])                     #uninitialized
```

`numpy.`**`empty_like`**(*prototype*, *dtype=None*, *order='K'*, *subok=True*, *shape=None*)

Return a new array with the same shape and type as a given array.

**Parameters**

**prototype**

[array_like] The shape and data-type of *prototype* define these same attributes of the returned array.

**dtype**

[data-type, optional] Overrides the data type of the result.

New in version 1.6.0.

**order**

[{'C', 'F', 'A', or 'K'}, optional] Overrides the memory layout of the result. 'C' means C-order, 'F' means F-order, 'A' means 'F' if `prototype` is Fortran contiguous, 'C' otherwise. 'K' means match the layout of `prototype` as closely as possible.

New in version 1.6.0.

**subok**

[bool, optional.] If True, then the newly created array will use the sub-class type of 'a', otherwise it will be a base-class array. Defaults to True.

**shape**

[int or sequence of ints, optional.] Overrides the shape of the result. If order='K' and the number of dimensions is unchanged, will try to keep order, otherwise, order='C' is implied.

New in version 1.17.0.

**Returns**

**out**

[ndarray] Array of uninitialized (arbitrary) data with the same shape and type as *prototype*.

**See also:**

***`ones_like`***

Return an array of ones with shape and type of input.

***`zeros_like`***

Return an array of zeros with shape and type of input.

***`full_like`***

Return a new array with shape of input filled with value.

***`empty`***

Return a new uninitialized array.

### Notes

This function does *not* initialize the returned array; to do that use `zeros_like` or `ones_like` instead. It may be marginally faster than the functions that do set the array values.

### Examples

```
>>> a = ([1,2,3], [4,5,6])                         # a is array-like
>>> np.empty_like(a)
array([[-1073741821, -1073741821,           3],    # uninitialized
       [          0,           0, -1073741821]])
>>> a = np.array([[1., 2., 3.],[4.,5.,6.]])
>>> np.empty_like(a)
array([[ -2.00000715e+000,   1.48219694e-323,  -2.00000572e+000], # uninitialized
       [  4.38791518e-305,  -2.00000715e+000,   4.17269252e-309]])
```

`numpy.`**`eye`**(*N*, *M=None*, *k=0*, *dtype=<class 'float'>*, *order='C'*)

Return a 2-D array with ones on the diagonal and zeros elsewhere.

**Parameters**

**N**

[int] Number of rows in the output.

**M**

[int, optional] Number of columns in the output. If None, defaults to *N*.

**k**

[int, optional] Index of the diagonal: 0 (the default) refers to the main diagonal, a positive value refers to an upper diagonal, and a negative value to a lower diagonal.

**dtype**

[data-type, optional] Data-type of the returned array.

**order**

[{'C', 'F'}, optional] Whether the output should be stored in row-major (C-style) or column-major (Fortran-style) order in memory.

New in version 1.14.0.

**Returns**

**I**

[ndarray of shape (N,M)] An array where all elements are equal to zero, except for the *k*-th diagonal, whose values are equal to one.

**See also:**

**`identity`**

(almost) equivalent function

**`diag`**

diagonal 2-D array from a 1-D array specified by the user.

## Examples

```
>>> np.eye(2, dtype=int)
array([[1, 0],
       [0, 1]])
>>> np.eye(3, k=1)
array([[0.,  1.,  0.],
       [0.,  0.,  1.],
       [0.,  0.,  0.]])
```

`numpy.`**`identity`**(*n*, *dtype=None*)

Return the identity array.

The identity array is a square array with ones on the main diagonal.

**Parameters**

**n**

[int] Number of rows (and columns) in *n* x *n* output.

**dtype**

[data-type, optional] Data-type of the output. Defaults to `float`.

**Returns**

**out**

[ndarray] *n* x *n* array with its main diagonal set to one, and all other elements 0.

## Examples

```
>>> np.identity(3)
array([[1.,  0.,  0.],
       [0.,  1.,  0.],
       [0.,  0.,  1.]])
```

`numpy.`**`ones`**(*shape*, *dtype=None*, *order='C'*)

Return a new array of given shape and type, filled with ones.

**Parameters**

**shape**

[int or sequence of ints] Shape of the new array, e.g., `(2, 3)` or `2`.

**dtype**

[data-type, optional] The desired data-type for the array, e.g., `numpy.int8`. Default is `numpy.float64`.

**order**

[{'C', 'F'}, optional, default: C] Whether to store multi-dimensional data in row-major (C-style) or column-major (Fortran-style) order in memory.

**Returns**

**out**

[ndarray] Array of ones with the given shape, dtype, and order.

**See also:**

***`ones_like`***

Return an array of ones with shape and type of input.

***`empty`***

Return a new uninitialized array.

***`zeros`***

Return a new array setting values to zero.

***`full`***

Return a new array of given shape filled with value.

## Examples

```
>>> np.ones(5)
array([1., 1., 1., 1., 1.])
```

```
>>> np.ones((5,), dtype=int)
array([1, 1, 1, 1, 1])
```

```
>>> np.ones((2, 1))
array([[1.],
       [1.]])
```

```
>>> s = (2,2)
>>> np.ones(s)
array([[1.,  1.],
       [1.,  1.]])
```

`numpy.`**`ones_like`**(*a*, *dtype=None*, *order='K'*, *subok=True*, *shape=None*)

Return an array of ones with the same shape and type as a given array.

**Parameters**

**a**

[array_like] The shape and data-type of *a* define these same attributes of the returned array.

**dtype**

[data-type, optional] Overrides the data type of the result.

New in version 1.6.0.

**order**

[{'C', 'F', 'A', or 'K'}, optional] Overrides the memory layout of the result. 'C' means C-order, 'F' means F-order, 'A' means 'F' if *a* is Fortran contiguous, 'C' otherwise. 'K' means match the layout of *a* as closely as possible.

New in version 1.6.0.

**subok**

[bool, optional.] If True, then the newly created array will use the sub-class type of 'a', otherwise it will be a base-class array. Defaults to True.

**shape**

[int or sequence of ints, optional.] Overrides the shape of the result. If order='K' and the number of dimensions is unchanged, will try to keep order, otherwise, order='C' is implied.

New in version 1.17.0.

**Returns**

**out**

[ndarray] Array of ones with the same shape and type as *a*.

**See also:**

***`empty_like`***

Return an empty array with shape and type of input.

***`zeros_like`***

Return an array of zeros with shape and type of input.

***`full_like`***

Return a new array with shape of input filled with value.

***`ones`***

Return a new array setting values to one.

### Examples

```
>>> x = np.arange(6)
>>> x = x.reshape((2, 3))
>>> x
array([[0, 1, 2],
       [3, 4, 5]])
>>> np.ones_like(x)
array([[1, 1, 1],
       [1, 1, 1]])
```

```
>>> y = np.arange(3, dtype=float)
>>> y
array([0., 1., 2.])
>>> np.ones_like(y)
array([1.,  1.,  1.])
```

`numpy.`**`zeros`**(*shape*, *dtype=float*, *order='C'*)

Return a new array of given shape and type, filled with zeros.

**Parameters**

**shape**

[int or tuple of ints] Shape of the new array, e.g., `(2, 3)` or `2`.

**dtype**

[data-type, optional] The desired data-type for the array, e.g., `numpy.int8`. Default is `numpy.float64`.

**order**

[{'C', 'F'}, optional, default: 'C'] Whether to store multi-dimensional data in row-major (C-style) or column-major (Fortran-style) order in memory.

**Returns**

**out**

[ndarray] Array of zeros with the given shape, dtype, and order.

**See also:**

**`zeros_like`**

Return an array of zeros with shape and type of input.

**`empty`**

Return a new uninitialized array.

**`ones`**

Return a new array setting values to one.

**`full`**

Return a new array of given shape filled with value.

## Examples

```
>>> np.zeros(5)
array([ 0.,  0.,  0.,  0.,  0.])
```

```
>>> np.zeros((5,), dtype=int)
array([0, 0, 0, 0, 0])
```

```
>>> np.zeros((2, 1))
array([[ 0.],
       [ 0.]])
```

```
>>> s = (2,2)
>>> np.zeros(s)
array([[ 0.,  0.],
       [ 0.,  0.]])
```

```
>>> np.zeros((2,), dtype=[('x', 'i4'), ('y', 'i4')]) # custom dtype
array([(0, 0), (0, 0)],
      dtype=[('x', '<i4'), ('y', '<i4')])
```

`numpy.`**`zeros_like`**(*a*, *dtype=None*, *order='K'*, *subok=True*, *shape=None*)

Return an array of zeros with the same shape and type as a given array.

**Parameters**

**a**

[array_like] The shape and data-type of *a* define these same attributes of the returned array.

**dtype**

[data-type, optional] Overrides the data type of the result.

New in version 1.6.0.

**order**

[{'C', 'F', 'A', or 'K'}, optional] Overrides the memory layout of the result. 'C' means C-order, 'F' means F-order, 'A' means 'F' if *a* is Fortran contiguous, 'C' otherwise. 'K' means match the layout of *a* as closely as possible.

New in version 1.6.0.

**subok**

[bool, optional.] If True, then the newly created array will use the sub-class type of 'a', otherwise it will be a base-class array. Defaults to True.

**shape**

[int or sequence of ints, optional.] Overrides the shape of the result. If order='K' and the number of dimensions is unchanged, will try to keep order, otherwise, order='C' is implied.

New in version 1.17.0.

**Returns**

**out**

[ndarray] Array of zeros with the same shape and type as *a*.

**See also:**

*`empty_like`*

Return an empty array with shape and type of input.

*`ones_like`*

Return an array of ones with shape and type of input.

*`full_like`*

Return a new array with shape of input filled with value.

*`zeros`*

Return a new array setting values to zero.

## Examples

```
>>> x = np.arange(6)
>>> x = x.reshape((2, 3))
>>> x
array([[0, 1, 2],
       [3, 4, 5]])
>>> np.zeros_like(x)
array([[0, 0, 0],
       [0, 0, 0]])
```

```
>>> y = np.arange(3, dtype=float)
>>> y
array([0., 1., 2.])
>>> np.zeros_like(y)
array([0.,  0.,  0.])
```

`numpy.`**`full`**`(`*shape*, *fill_value*, *dtype=None*, *order='C'*`)`

Return a new array of given shape and type, filled with *fill_value*.

**Parameters**

**shape**

[int or sequence of ints] Shape of the new array, e.g., `(2, 3)` or `2`.

**fill_value**

[scalar or array_like] Fill value.

**dtype**

[data-type, optional]

**The desired data-type for the array The default, None, means**

*np.array(fill_value).dtype*.

**order**

[{'C', 'F'}, optional] Whether to store multidimensional data in C- or Fortran-contiguous (row- or column-wise) order in memory.

**Returns**

**out**

[ndarray] Array of *fill_value* with the given shape, dtype, and order.

**See also:**

***`full_like`***

Return a new array with shape of input filled with value.

***`empty`***

Return a new uninitialized array.

***`ones`***

Return a new array setting values to one.

*`zeros`*

Return a new array setting values to zero.

### Examples

```
>>> np.full((2, 2), np.inf)
array([[inf, inf],
       [inf, inf]])
>>> np.full((2, 2), 10)
array([[10, 10],
       [10, 10]])
```

```
>>> np.full((2, 2), [1, 2])
array([[1, 2],
       [1, 2]])
```

`numpy.`**`full_like`**(*a*, *fill_value*, *dtype=None*, *order='K'*, *subok=True*, *shape=None*)

Return a full array with the same shape and type as a given array.

**Parameters**

**a**

[array_like] The shape and data-type of *a* define these same attributes of the returned array.

**fill_value**

[scalar] Fill value.

**dtype**

[data-type, optional] Overrides the data type of the result.

**order**

[{'C', 'F', 'A', or 'K'}, optional] Overrides the memory layout of the result. 'C' means C-order, 'F' means F-order, 'A' means 'F' if *a* is Fortran contiguous, 'C' otherwise. 'K' means match the layout of *a* as closely as possible.

**subok**

[bool, optional.] If True, then the newly created array will use the sub-class type of 'a', otherwise it will be a base-class array. Defaults to True.

**shape**

[int or sequence of ints, optional.] Overrides the shape of the result. If order='K' and the number of dimensions is unchanged, will try to keep order, otherwise, order='C' is implied.

New in version 1.17.0.

**Returns**

**out**

[ndarray] Array of *fill_value* with the same shape and type as *a*.

**See also:**

**`empty_like`**

Return an empty array with shape and type of input.

**`ones_like`**

Return an array of ones with shape and type of input.

**`zeros_like`**

Return an array of zeros with shape and type of input.

**`full`**

Return a new array of given shape filled with value.

### Examples

```
>>> x = np.arange(6, dtype=int)
>>> np.full_like(x, 1)
array([1, 1, 1, 1, 1, 1])
>>> np.full_like(x, 0.1)
array([0, 0, 0, 0, 0, 0])
>>> np.full_like(x, 0.1, dtype=np.double)
array([0.1, 0.1, 0.1, 0.1, 0.1, 0.1])
>>> np.full_like(x, np.nan, dtype=np.double)
array([nan, nan, nan, nan, nan, nan])
```

```
>>> y = np.arange(6, dtype=np.double)
>>> np.full_like(y, 0.1)
array([0.1,  0.1,  0.1,  0.1,  0.1,  0.1])
```

## 4.1.2 From existing data

| | |
|---|---|
| `array`(object[, dtype, copy, order, subok, ndmin]) | Create an array. |
| `asarray`(a[, dtype, order]) | Convert the input to an array. |
| `asanyarray`(a[, dtype, order]) | Convert the input to an ndarray, but pass ndarray subclasses through. |
| `ascontiguousarray`(a[, dtype]) | Return a contiguous array (ndim >= 1) in memory (C order). |
| `asmatrix`(data[, dtype]) | Interpret the input as a matrix. |
| `copy`(a[, order, subok]) | Return an array copy of the given object. |
| `frombuffer`(buffer[, dtype, count, offset]) | Interpret a buffer as a 1-dimensional array. |
| `fromfile`(file[, dtype, count, sep, offset]) | Construct an array from data in a text or binary file. |
| `fromfunction`(function, shape, *[, dtype]) | Construct an array by executing a function over each coordinate. |
| `fromiter`(iterable, dtype[, count]) | Create a new 1-dimensional array from an iterable object. |
| `fromstring`(string[, dtype, count, sep]) | A new 1-D array initialized from text data in a string. |
| `loadtxt`(fname[, dtype, comments, delimiter, …]) | Load data from a text file. |

`numpy.`**`array`**(*object, dtype=None, *, copy=True, order='K', subok=False, ndmin=0*)

Create an array.

**Parameters**

**object**

[array_like] An array, any object exposing the array interface, an object whose __array__ method returns an array, or any (nested) sequence.

**dtype**

[data-type, optional] The desired data-type for the array. If not given, then the type will be determined as the minimum type required to hold the objects in the sequence.

**copy**

[bool, optional] If true (default), then the object is copied. Otherwise, a copy will only be made if __array__ returns a copy, if obj is a nested sequence, or if a copy is needed to satisfy any of the other requirements (*`dtype`*, *order*, etc.).

**order**

[{'K', 'A', 'C', 'F'}, optional] Specify the memory layout of the array. If object is not an array, the newly created array will be in C order (row major) unless 'F' is specified, in which case it will be in Fortran order (column major). If object is an array the following holds.

| order | no copy | copy=True |
|---|---|---|
| 'K' | unchanged | F & C order preserved, otherwise most similar order |
| 'A' | unchanged | F order if input is F and not C, otherwise C order |
| 'C' | C order | C order |
| 'F' | F order | F order |

When `copy=False` and a copy is made for other reasons, the result is the same as if `copy=True`, with some exceptions for *A*, see the Notes section. The default order is 'K'.

**subok**

[bool, optional] If True, then sub-classes will be passed-through, otherwise the returned array will be forced to be a base-class array (default).

**ndmin**

[int, optional] Specifies the minimum number of dimensions that the resulting array should have. Ones will be pre-pended to the shape as needed to meet this requirement.

**Returns**

**out**

[ndarray] An array object satisfying the specified requirements.

**See also:**

***`empty_like`***

Return an empty array with shape and type of input.

***`ones_like`***

Return an array of ones with shape and type of input.

***`zeros_like`***

Return an array of zeros with shape and type of input.

***full_like***

Return a new array with shape of input filled with value.

***empty***

Return a new uninitialized array.

***ones***

Return a new array setting values to one.

***zeros***

Return a new array setting values to zero.

***full***

Return a new array of given shape filled with value.

### Notes

When order is 'A' and `object` is an array in neither 'C' nor 'F' order, and a copy is forced by a change in dtype, then the order of the result is not necessarily 'C' as expected. This is likely a bug.

### Examples

```
>>> np.array([1, 2, 3])
array([1, 2, 3])
```

Upcasting:

```
>>> np.array([1, 2, 3.0])
array([ 1.,  2.,  3.])
```

More than one dimension:

```
>>> np.array([[1, 2], [3, 4]])
array([[1, 2],
       [3, 4]])
```

Minimum dimensions 2:

```
>>> np.array([1, 2, 3], ndmin=2)
array([[1, 2, 3]])
```

Type provided:

```
>>> np.array([1, 2, 3], dtype=complex)
array([ 1.+0.j,  2.+0.j,  3.+0.j])
```

Data-type consisting of more than one element:

```
>>> x = np.array([(1,2),(3,4)],dtype=[('a','<i4'),('b','<i4')])
>>> x['a']
array([1, 3])
```

Creating an array from sub-classes:

```
>>> np.array(np.mat('1 2; 3 4'))
array([[1, 2],
       [3, 4]])
```

```
>>> np.array(np.mat('1 2; 3 4'), subok=True)
matrix([[1, 2],
        [3, 4]])
```

`numpy.`**`asarray`**(*a*, *dtype=None*, *order=None*)

Convert the input to an array.

**Parameters**

**a**

[array_like] Input data, in any form that can be converted to an array. This includes lists, lists of tuples, tuples, tuples of tuples, tuples of lists and ndarrays.

**dtype**

[data-type, optional] By default, the data-type is inferred from the input data.

**order**

[{'C', 'F'}, optional] Whether to use row-major (C-style) or column-major (Fortran-style) memory representation. Defaults to 'C'.

**Returns**

**out**

[ndarray] Array interpretation of *a*. No copy is performed if the input is already an ndarray with matching dtype and order. If *a* is a subclass of ndarray, a base class ndarray is returned.

**See also:**

***`asanyarray`***

Similar function which passes through subclasses.

***`ascontiguousarray`***

Convert input to a contiguous array.

***`asfarray`***

Convert input to a floating point ndarray.

***`asfortranarray`***

Convert input to an ndarray with column-major memory order.

***`asarray_chkfinite`***

Similar function which checks input for NaNs and Infs.

***`fromiter`***

Create an array from an iterator.

***`fromfunction`***

Construct an array by executing a function on grid positions.

## Examples

Convert a list into an array:

```
>>> a = [1, 2]
>>> np.asarray(a)
array([1, 2])
```

Existing arrays are not copied:

```
>>> a = np.array([1, 2])
>>> np.asarray(a) is a
True
```

If `dtype` is set, array is copied only if dtype does not match:

```
>>> a = np.array([1, 2], dtype=np.float32)
>>> np.asarray(a, dtype=np.float32) is a
True
>>> np.asarray(a, dtype=np.float64) is a
False
```

Contrary to `asanyarray`, ndarray subclasses are not passed through:

```
>>> issubclass(np.recarray, np.ndarray)
True
>>> a = np.array([(1.0, 2), (3.0, 4)], dtype='f4,i4').view(np.recarray)
>>> np.asarray(a) is a
False
>>> np.asanyarray(a) is a
True
```

`numpy.`**`asanyarray`**(*a*, *dtype=None*, *order=None*)

Convert the input to an ndarray, but pass ndarray subclasses through.

**Parameters**

**a**

[array_like] Input data, in any form that can be converted to an array. This includes scalars, lists, lists of tuples, tuples, tuples of tuples, tuples of lists, and ndarrays.

**dtype**

[data-type, optional] By default, the data-type is inferred from the input data.

**order**

[{‘C’, ‘F’}, optional] Whether to use row-major (C-style) or column-major (Fortran-style) memory representation. Defaults to ‘C’.

**Returns**

**out**

[ndarray or an ndarray subclass] Array interpretation of *a*. If *a* is an ndarray or a subclass of ndarray, it is returned as-is and no copy is performed.

**See also:**

***`asarray`***

Similar function which always returns ndarrays.

***`ascontiguousarray`***

Convert input to a contiguous array.

***`asfarray`***

Convert input to a floating point ndarray.

***`asfortranarray`***

Convert input to an ndarray with column-major memory order.

***`asarray_chkfinite`***

Similar function which checks input for NaNs and Infs.

***`fromiter`***

Create an array from an iterator.

***`fromfunction`***

Construct an array by executing a function on grid positions.

### Examples

Convert a list into an array:

```
>>> a = [1, 2]
>>> np.asanyarray(a)
array([1, 2])
```

Instances of *`ndarray`* subclasses are passed through as-is:

```
>>> a = np.array([(1.0, 2), (3.0, 4)], dtype='f4,i4').view(np.recarray)
>>> np.asanyarray(a) is a
True
```

`numpy.`**`ascontiguousarray`**(*a*, *dtype=None*)

Return a contiguous array (ndim >= 1) in memory (C order).

**Parameters**

**a**

[array_like] Input array.

**dtype**

[str or dtype object, optional] Data-type of returned array.

**Returns**

**out**

[ndarray] Contiguous array of same shape and content as *a*, with type *`dtype`* if specified.

**See also:**

***asfortranarray***

Convert input to an ndarray with column-major memory order.

***require***

Return an ndarray that satisfies requirements.

***ndarray.flags***

Information about the memory layout of the array.

### Examples

```
>>> x = np.arange(6).reshape(2,3)
>>> np.ascontiguousarray(x, dtype=np.float32)
array([[0., 1., 2.],
       [3., 4., 5.]], dtype=float32)
>>> x.flags['C_CONTIGUOUS']
True
```

Note: This function returns an array with at least one-dimension (1-d) so it will not preserve 0-d arrays.

`numpy.`**`copy`**(*a*, *order='K'*, *subok=False*)

Return an array copy of the given object.

**Parameters**

**a**

[array_like] Input data.

**order**

[{'C', 'F', 'A', 'K'}, optional] Controls the memory layout of the copy. 'C' means C-order, 'F' means F-order, 'A' means 'F' if *a* is Fortran contiguous, 'C' otherwise. 'K' means match the layout of *a* as closely as possible. (Note that this function and `ndarray.copy` are very similar, but have different default values for their order= arguments.)

**subok**

[bool, optional] If True, then sub-classes will be passed-through, otherwise the returned array will be forced to be a base-class array (defaults to False).

New in version 1.19.0.

**Returns**

**arr**

[ndarray] Array interpretation of *a*.

**See also:**

***ndarray.copy***

Preferred method for creating an array copy

## Notes

This is equivalent to:

```
>>> np.array(a, copy=True)
```

## Examples

Create an array x, with a reference y and a copy z:

```
>>> x = np.array([1, 2, 3])
>>> y = x
>>> z = np.copy(x)
```

Note that, when we modify x, y changes, but not z:

```
>>> x[0] = 10
>>> x[0] == y[0]
True
>>> x[0] == z[0]
False
```

Note that np.copy is a shallow copy and will not copy object elements within arrays. This is mainly important for arrays containing Python objects. The new array will contain the same object which may lead to surprises if that object can be modified (is mutable):

```
>>> a = np.array([1, 'm', [2, 3, 4]], dtype=object)
>>> b = np.copy(a)
>>> b[2][0] = 10
>>> a
array([1, 'm', list([10, 3, 4])], dtype=object)
```

To ensure all elements within an `object` array are copied, use `copy.deepcopy`:

```
>>> import copy
>>> a = np.array([1, 'm', [2, 3, 4]], dtype=object)
>>> c = copy.deepcopy(a)
>>> c[2][0] = 10
>>> c
array([1, 'm', list([10, 3, 4])], dtype=object)
>>> a
array([1, 'm', list([2, 3, 4])], dtype=object)
```

`numpy.`**`frombuffer`**(*buffer*, *dtype=float*, *count=-1*, *offset=0*)

Interpret a buffer as a 1-dimensional array.

**Parameters**

**buffer**

[buffer_like] An object that exposes the buffer interface.

**dtype**

[data-type, optional] Data-type of the returned array; default: float.

**count**

[int, optional] Number of items to read. `-1` means all data in the buffer.

**offset**

[int, optional] Start reading the buffer from this offset (in bytes); default: 0.

## Notes

If the buffer has data that is not in machine byte-order, this should be specified as part of the data-type, e.g.:

```
>>> dt = np.dtype(int)
>>> dt = dt.newbyteorder('>')
>>> np.frombuffer(buf, dtype=dt)
```

The data of the resulting array will not be byteswapped, but will be interpreted correctly.

## Examples

```
>>> s = b'hello world'
>>> np.frombuffer(s, dtype='S1', count=5, offset=6)
array([b'w', b'o', b'r', b'l', b'd'], dtype='|S1')
```

```
>>> np.frombuffer(b'\x01\x02', dtype=np.uint8)
array([1, 2], dtype=uint8)
>>> np.frombuffer(b'\x01\x02\x03\x04\x05', dtype=np.uint8, count=3)
array([1, 2, 3], dtype=uint8)
```

`numpy.`**`fromfile`**(*file*, *dtype=float*, *count=-1*, *sep=''*, *offset=0*)

Construct an array from data in a text or binary file.

A highly efficient way of reading binary data with a known data-type, as well as parsing simply formatted text files. Data written using the *tofile* method can be read using this function.

**Parameters**

**file**

[file or str or Path] Open file object or filename.

Changed in version 1.17.0: `pathlib.Path` objects are now accepted.

**dtype**

[data-type] Data type of the returned array. For binary files, it is used to determine the size and byte-order of the items in the file. Most builtin numeric types are supported and extension types may be supported.

New in version 1.18.0: Complex dtypes.

**count**

[int] Number of items to read. `-1` means all items (i.e., the complete file).

**sep**

[str] Separator between items if file is a text file. Empty ("") separator means the file should be treated as binary. Spaces (" ") in the separator match zero or more whitespace characters. A separator consisting only of spaces must match at least one whitespace.

**offset**

[int] The offset (in bytes) from the file's current position. Defaults to 0. Only permitted for binary files.

New in version 1.17.0.

**See also:**

`load`, `save`, `ndarray.tofile`

**`loadtxt`**

More flexible way of loading data from a text file.

### Notes

Do not rely on the combination of *tofile* and `fromfile` for data storage, as the binary files generated are not platform independent. In particular, no byte-order or data-type information is saved. Data can be stored in the platform independent `.npy` format using `save` and `load` instead.

### Examples

Construct an ndarray:

```
>>> dt = np.dtype([('time', [('min', np.int64), ('sec', np.int64)]),
...                ('temp', float)])
>>> x = np.zeros((1,), dtype=dt)
>>> x['time']['min'] = 10; x['temp'] = 98.25
>>> x
array([((10, 0), 98.25)],
      dtype=[('time', [('min', '<i8'), ('sec', '<i8')]), ('temp', '<f8')])
```

Save the raw data to disk:

```
>>> import tempfile
>>> fname = tempfile.mkstemp()[1]
>>> x.tofile(fname)
```

Read the raw data from disk:

```
>>> np.fromfile(fname, dtype=dt)
array([((10, 0), 98.25)],
      dtype=[('time', [('min', '<i8'), ('sec', '<i8')]), ('temp', '<f8')])
```

The recommended way to store and load data:

```
>>> np.save(fname, x)
>>> np.load(fname + '.npy')
array([((10, 0), 98.25)],
      dtype=[('time', [('min', '<i8'), ('sec', '<i8')]), ('temp', '<f8')])
```

`numpy.`**`fromfunction`**(*function*, *shape*, *, *dtype=<class 'float'>*, ***kwargs*)

Construct an array by executing a function over each coordinate.

The resulting array therefore has a value `fn(x, y, z)` at coordinate `(x, y, z)`.

**Parameters**

**function**

[callable] The function is called with N parameters, where N is the rank of `shape`. Each parameter represents the coordinates of the array varying along a specific axis. For example, if `shape` were `(2, 2)`, then the parameters would be `array([[0, 0], [1, 1]])` and `array([[0, 1], [0, 1]])`

**shape**

[(N,) tuple of ints] Shape of the output array, which also determines the shape of the coordinate arrays passed to *function*.

**dtype**

[data-type, optional] Data-type of the coordinate arrays passed to *function*. By default, `dtype` is float.

**Returns**

**fromfunction**

[any] The result of the call to *function* is passed back directly. Therefore the shape of `fromfunction` is completely determined by *function*. If *function* returns a scalar value, the shape of `fromfunction` would not match the `shape` parameter.

**See also:**

`indices`, `meshgrid`

## Notes

Keywords other than `dtype` are passed to *function*.

## Examples

```
>>> np.fromfunction(lambda i, j: i == j, (3, 3), dtype=int)
array([[ True, False, False],
       [False,  True, False],
       [False, False,  True]])
```

```
>>> np.fromfunction(lambda i, j: i + j, (3, 3), dtype=int)
array([[0, 1, 2],
       [1, 2, 3],
       [2, 3, 4]])
```

`numpy.`**`fromiter`**(*iterable*, *dtype*, *count=-1*)

Create a new 1-dimensional array from an iterable object.

**Parameters**

**iterable**

[iterable object] An iterable object providing data for the array.

**dtype**

[data-type] The data-type of the returned array.

**count**

[int, optional] The number of items to read from *iterable*. The default is -1, which means all data is read.

**Returns**

**out**

[ndarray] The output array.

## Notes

Specify *count* to improve performance. It allows `fromiter` to pre-allocate the output array, instead of resizing it on demand.

## Examples

```
>>> iterable = (x*x for x in range(5))
>>> np.fromiter(iterable, float)
array([  0.,   1.,   4.,   9.,  16.])
```

`numpy.`**`fromstring`**(*string*, *dtype=float*, *count=-1*, *sep=''*)

A new 1-D array initialized from text data in a string.

**Parameters**

**string**

[str] A string containing the data.

**dtype**

[data-type, optional] The data type of the array; default: float. For binary input data, the data must be in exactly this format. Most builtin numeric types are supported and extension types may be supported.

New in version 1.18.0: Complex dtypes.

**count**

[int, optional] Read this number of `dtype` elements from the data. If this is negative (the default), the count will be determined from the length of the data.

**sep**

[str, optional] The string separating numbers in the data; extra whitespace between elements is also ignored.

Deprecated since version 1.14: Passing `sep=''`, the default, is deprecated since it will trigger the deprecated binary mode of this function. This mode interprets `string` as binary bytes, rather than ASCII text with decimal numbers, an operation which is better spelt `frombuffer(string, dtype, count)`. If `string` contains unicode text, the binary mode of `fromstring` will first encode it into bytes using either utf-8 (python 3) or the default encoding (python 2), neither of which produce sane results.

**Returns**

**arr**

[ndarray] The constructed array.

**Raises**

**ValueError**

If the string is not the correct size to satisfy the requested `dtype` and *count*.

**See also:**

`frombuffer`, `fromfile`, `fromiter`

### Examples

```
>>> np.fromstring('1 2', dtype=int, sep=' ')
array([1, 2])
>>> np.fromstring('1, 2', dtype=int, sep=',')
array([1, 2])
```

`numpy.loadtxt`(*fname*, *dtype=<class 'float'>*, *comments='#'*, *delimiter=None*, *converters=None*, *skiprows=0*, *usecols=None*, *unpack=False*, *ndmin=0*, *encoding='bytes'*, *max_rows=None*)

Load data from a text file.

Each row in the text file must have the same number of values.

**Parameters**

**fname**

[file, str, or pathlib.Path] File, filename, or generator to read. If the filename extension is `.gz` or `.bz2`, the file is first decompressed. Note that generators should return byte strings.

**dtype**

[data-type, optional] Data-type of the resulting array; default: float. If this is a structured data-type, the resulting array will be 1-dimensional, and each row will be interpreted as an element of the array. In this case, the number of columns used must match the number of fields in the data-type.

**comments**

[str or sequence of str, optional] The characters or list of characters used to indicate the start of a comment. None implies no comments. For backwards compatibility, byte strings will be decoded as 'latin1'. The default is '#'.

**delimiter**

[str, optional] The string used to separate values. For backwards compatibility, byte strings will be decoded as 'latin1'. The default is whitespace.

**converters**

[dict, optional] A dictionary mapping column number to a function that will parse the column string into the desired value. E.g., if column 0 is a date string: `converters = {0: datestr2num}`. Converters can also be used to provide a default value for missing data (but see also `genfromtxt`): `converters = {3: lambda s: float(s.strip() or 0)}`. Default: None.

**skiprows**

[int, optional] Skip the first *skiprows* lines, including comments; default: 0.

**usecols**

[int or sequence, optional] Which columns to read, with 0 being the first. For example, `usecols = (1,4,5)` will extract the 2nd, 5th and 6th columns. The default, None, results in all columns being read.

Changed in version 1.11.0: When a single column has to be read it is possible to use an integer instead of a tuple. E.g `usecols = 3` reads the fourth column the same way as `usecols = (3,)` would.

**unpack**

[bool, optional] If True, the returned array is transposed, so that arguments may be unpacked using `x, y, z = loadtxt(...)`. When used with a structured data-type, arrays are returned for each field. Default is False.

**ndmin**

[int, optional] The returned array will have at least *ndmin* dimensions. Otherwise mono-dimensional axes will be squeezed. Legal values: 0 (default), 1 or 2.

New in version 1.6.0.

**encoding**

[str, optional] Encoding used to decode the inputfile. Does not apply to input streams. The special value 'bytes' enables backward compatibility workarounds that ensures you receive byte arrays as results if possible and passes 'latin1' encoded strings to converters. Override this value to receive unicode arrays and pass strings as input to converters. If set to None the system default is used. The default value is 'bytes'.

New in version 1.14.0.

**max_rows**

[int, optional] Read *max_rows* lines of content after *skiprows* lines. The default is to read all the lines.

New in version 1.16.0.

**Returns**

**out**

[ndarray] Data read from the text file.

**See also:**

`load`, `fromstring`, `fromregex`

**`genfromtxt`**

Load data with missing values handled as specified.

**`scipy.io.loadmat`**

reads MATLAB data files

### Notes

This function aims to be a fast reader for simply formatted files. The `genfromtxt` function provides more sophisticated handling of, e.g., lines with missing values.

New in version 1.10.0.

The strings produced by the Python float.hex method can be used as input for floats.

### Examples

```
>>> from io import StringIO   # StringIO behaves like a file object
>>> c = StringIO("0 1\n2 3")
>>> np.loadtxt(c)
array([[0., 1.],
       [2., 3.]])
```

```
>>> d = StringIO("M 21 72\nF 35 58")
>>> np.loadtxt(d, dtype={'names': ('gender', 'age', 'weight'),
...                      'formats': ('S1', 'i4', 'f4')})
array([(b'M', 21, 72.), (b'F', 35, 58.)],
      dtype=[('gender', 'S1'), ('age', '<i4'), ('weight', '<f4')])
```

```
>>> c = StringIO("1,0,2\n3,0,4")
>>> x, y = np.loadtxt(c, delimiter=',', usecols=(0, 2), unpack=True)
>>> x
array([1., 3.])
>>> y
array([2., 4.])
```

This example shows how *converters* can be used to convert a field with a trailing minus sign into a negative number.

```
>>> s = StringIO('10.01 31.25-\n19.22 64.31\n17.57- 63.94')
>>> def conv(fld):
...     return -float(fld[:-1]) if fld.endswith(b'-') else float(fld)
...
>>> np.loadtxt(s, converters={0: conv, 1: conv})
array([[ 10.01, -31.25],
       [ 19.22,  64.31],
       [-17.57,  63.94]])
```

## 4.1.3 Creating record arrays (`numpy.rec`)

**Note:** `numpy.rec` is the preferred alias for `numpy.core.records`.

| | |
|---|---|
| `core.records.array`(obj[, dtype, shape, …]) | Construct a record array from a wide-variety of objects. |
| `core.records.fromarrays`(arrayList[, dtype, …]) | Create a record array from a (flat) list of arrays |
| `core.records.fromrecords`(recList[, dtype, …]) | Create a recarray from a list of records in text form. |

Continued on next page

Table 3 – continued from previous page

| | |
|---|---|
| `core.records.fromstring`(datastring[, dtype, …]) | Create a record array from binary data |
| `core.records.fromfile`(fd[, dtype, shape, …]) | Create an array from binary file data |

numpy.core.records.**array**(*obj*, *dtype=None*, *shape=None*, *offset=0*, *strides=None*, *formats=None*, *names=None*, *titles=None*, *aligned=False*, *byteorder=None*, *copy=True*)

Construct a record array from a wide-variety of objects.

numpy.core.records.**fromarrays**(*arrayList*, *dtype=None*, *shape=None*, *formats=None*, *names=None*, *titles=None*, *aligned=False*, *byteorder=None*)

Create a record array from a (flat) list of arrays

**Parameters**

**arrayList**

[list or tuple] List of array-like objects (such as lists, tuples, and ndarrays).

**dtype**

[data-type, optional] valid dtype for all arrays

**shape**

[int or tuple of ints, optional] Shape of the resulting array. If not provided, inferred from `arrayList[0]`.

**formats, names, titles, aligned, byteorder :**

If *dtype* is `None`, these arguments are passed to `numpy.format_parser` to construct a dtype. See that function for detailed documentation.

**Returns**

**np.recarray**

Record array consisting of given arrayList columns.

### Examples

```
>>> x1=np.array([1,2,3,4])
>>> x2=np.array(['a','dd','xyz','12'])
>>> x3=np.array([1.1,2,3,4])
>>> r = np.core.records.fromarrays([x1,x2,x3],names='a,b,c')
>>> print(r[1])
(2, 'dd', 2.0) # may vary
>>> x1[1]=34
>>> r.a
array([1, 2, 3, 4])
```

```
>>> x1 = np.array([1, 2, 3, 4])
>>> x2 = np.array(['a', 'dd', 'xyz', '12'])
>>> x3 = np.array([1.1, 2, 3,4])
>>> r = np.core.records.fromarrays(
...     [x1, x2, x3],
...     dtype=np.dtype([('a', np.int32), ('b', 'S3'), ('c', np.float32)]))
>>> r
```

(continues on next page)

(continued from previous page)

```
rec.array([(1, b'a', 1.1), (2, b'dd', 2. ), (3, b'xyz', 3. ),
           (4, b'12', 4. )],
          dtype=[('a', '<i4'), ('b', 'S3'), ('c', '<f4')])
```

`numpy.core.records.`**`fromrecords`**(*recList*, *dtype=None*, *shape=None*, *formats=None*, *names=None*, *titles=None*, *aligned=False*, *byteorder=None*)

Create a recarray from a list of records in text form.

**Parameters**

**recList**

[sequence] data in the same field may be heterogeneous - they will be promoted to the highest data type.

**dtype**

[data-type, optional] valid dtype for all arrays

**shape**

[int or tuple of ints, optional] shape of each array.

**formats, names, titles, aligned, byteorder :**

If *dtype* is `None`, these arguments are passed to *`numpy.format_parser`* to construct a dtype. See that function for detailed documentation.

If both *formats* and *dtype* are None, then this will auto-detect formats. Use list of tuples rather than list of lists for faster processing.

**Returns**

**np.recarray**

record array consisting of given recList rows.

### Examples

```
>>> r=np.core.records.fromrecords([(456,'dbe',1.2),(2,'de',1.3)],
... names='col1,col2,col3')
>>> print(r[0])
(456, 'dbe', 1.2)
>>> r.col1
array([456,   2])
>>> r.col2
array(['dbe', 'de'], dtype='<U3')
>>> import pickle
>>> pickle.loads(pickle.dumps(r))
rec.array([(456, 'dbe', 1.2), (  2, 'de', 1.3)],
          dtype=[('col1', '<i8'), ('col2', '<U3'), ('col3', '<f8')])
```

`numpy.core.records.`**`fromstring`**(*datastring*, *dtype=None*, *shape=None*, *offset=0*, *formats=None*, *names=None*, *titles=None*, *aligned=False*, *byteorder=None*)

Create a record array from binary data

Note that despite the name of this function it does not accept *str* instances.

**Parameters**

**datastring**

[bytes-like] Buffer of binary data

**dtype**

[data-type, optional] Valid dtype for all arrays

**shape**

[int or tuple of ints, optional] Shape of each array.

**offset**

[int, optional] Position in the buffer to start reading from.

**formats, names, titles, aligned, byteorder :**

If *dtype* is `None`, these arguments are passed to *`numpy.format_parser`* to construct a dtype. See that function for detailed documentation.

**Returns**

**np.recarray**

Record array view into the data in datastring. This will be readonly if *datastring* is readonly.

**See also:**

*`numpy.frombuffer`*

### Examples

```
>>> a = b'\x01\x02\x03abc'
>>> np.core.records.fromstring(a, dtype='u1,u1,u1,S3')
rec.array([(1, 2, 3, b'abc')],
        dtype=[('f0', 'u1'), ('f1', 'u1'), ('f2', 'u1'), ('f3', 'S3')])
```

```
>>> grades_dtype = [('Name', (np.str_, 10)), ('Marks', np.float64),
...                 ('GradeLevel', np.int32)]
>>> grades_array = np.array([('Sam', 33.3, 3), ('Mike', 44.4, 5),
...                         ('Aadi', 66.6, 6)], dtype=grades_dtype)
>>> np.core.records.fromstring(grades_array.tobytes(), dtype=grades_dtype)
rec.array([('Sam', 33.3, 3), ('Mike', 44.4, 5), ('Aadi', 66.6, 6)],
        dtype=[('Name', '<U10'), ('Marks', '<f8'), ('GradeLevel', '<i4')])
```

```
>>> s = '\x01\x02\x03abc'
>>> np.core.records.fromstring(s, dtype='u1,u1,u1,S3')
Traceback (most recent call last)
   ...
TypeError: a bytes-like object is required, not 'str'
```

`numpy.core.records.`**`fromfile`**(*fd*, *dtype=None*, *shape=None*, *offset=0*, *formats=None*, *names=None*, *titles=None*, *aligned=False*, *byteorder=None*)

Create an array from binary file data

**Parameters**

**fd**

[str or file type] If file is a string or a path-like object then that file is opened, else it is assumed to be a file object. The file object must support random access (i.e. it must have tell and seek methods).

**dtype**

[data-type, optional] valid dtype for all arrays

**shape**

[int or tuple of ints, optional] shape of each array.

**offset**

[int, optional] Position in the file to start reading from.

**formats, names, titles, aligned, byteorder :**

If *dtype* is `None`, these arguments are passed to *`numpy.format_parser`* to construct a dtype. See that function for detailed documentation

**Returns**

**np.recarray**

record array consisting of data enclosed in file.

### Examples

```
>>> from tempfile import TemporaryFile
>>> a = np.empty(10,dtype='f8,i4,a5')
>>> a[5] = (0.5,10,'abcde')
>>>
>>> fd=TemporaryFile()
>>> a = a.newbyteorder('<')
>>> a.tofile(fd)
>>>
>>> _ = fd.seek(0)
>>> r=np.core.records.fromfile(fd, formats='f8,i4,a5', shape=10,
... byteorder='<')
>>> print(r[5])
(0.5, 10, 'abcde')
>>> r.shape
(10,)
```

## 4.1.4 Creating character arrays (`numpy.char`)

**Note:** *`numpy.char`* is the preferred alias for `numpy.core.defchararray`.

| | |
|---|---|
| *`core.defchararray.array`*(obj[, itemsize, …]) | Create a *`chararray`*. |
| *`core.defchararray.asarray`*(obj[, itemsize, …]) | Convert the input to a *`chararray`*, copying the data only if necessary. |

`numpy.core.defchararray.`**`asarray`**(*obj*, *itemsize=None*, *unicode=None*, *order=None*)

Convert the input to a `chararray`, copying the data only if necessary.

Versus a regular NumPy array of type *str* or *unicode*, this class adds the following functionality:

1) values automatically have whitespace removed from the end when indexed
2) comparison operators automatically remove whitespace from the end when comparing values
3) vectorized string operations are provided as methods (e.g. *str.endswith*) and infix operators (e.g. `+`, `*`,"%")

**Parameters**

**obj**

[array of str or unicode-like]

**itemsize**

[int, optional] *itemsize* is the number of characters per scalar in the resulting array. If *itemsize* is None, and *obj* is an object array or a Python list, the *itemsize* will be automatically determined. If *itemsize* is provided and *obj* is of type str or unicode, then the *obj* string will be chunked into *itemsize* pieces.

**unicode**

[bool, optional] When true, the resulting `chararray` can contain Unicode characters, when false only 8-bit characters. If unicode is None and *obj* is one of the following:

- a `chararray`,
- an ndarray of type *str* or 'unicode'
- a Python str or unicode object,

then the unicode setting of the output array will be automatically determined.

**order**

[{'C', 'F'}, optional] Specify the order of the array. If order is 'C' (default), then the array will be in C-contiguous order (last-index varies the fastest). If order is 'F', then the returned array will be in Fortran-contiguous order (first-index varies the fastest).

### 4.1.5 Numerical ranges

| | |
|---|---|
| `arange`([start,] stop[, step,][, dtype]) | Return evenly spaced values within a given interval. |
| `linspace`(start, stop[, num, endpoint, …]) | Return evenly spaced numbers over a specified interval. |
| `logspace`(start, stop[, num, endpoint, base, …]) | Return numbers spaced evenly on a log scale. |
| `geomspace`(start, stop[, num, endpoint, …]) | Return numbers spaced evenly on a log scale (a geometric progression). |
| `meshgrid`(*xi[, copy, sparse, indexing]) | Return coordinate matrices from coordinate vectors. |
| `mgrid` | *nd_grid* instance which returns a dense multi-dimensional "meshgrid". |
| `ogrid` | *nd_grid* instance which returns an open multi-dimensional "meshgrid". |

`numpy.`**`arange`**([*start*], *stop*[, *step*], *dtype=None*)

Return evenly spaced values within a given interval.

Values are generated within the half-open interval `[start, stop)` (in other words, the interval including *start* but excluding *stop*). For integer arguments the function is equivalent to the Python built-in *range* function, but returns an ndarray rather than a list.

When using a non-integer step, such as 0.1, the results will often not be consistent. It is better to use `numpy.linspace` for these cases.

**Parameters**

**start**

[number, optional] Start of interval. The interval includes this value. The default start value is 0.

**stop**

[number] End of interval. The interval does not include this value, except in some cases where *step* is not an integer and floating point round-off affects the length of *out*.

**step**

[number, optional] Spacing between values. For any output *out*, this is the distance between two adjacent values, `out[i+1] - out[i]`. The default step size is 1. If *step* is specified as a position argument, *start* must also be given.

**dtype**

[dtype] The type of the output array. If `dtype` is not given, infer the data type from the other input arguments.

**Returns**

**arange**

[ndarray] Array of evenly spaced values.

For floating point arguments, the length of the result is `ceil((stop - start)/step)`. Because of floating point overflow, this rule may result in the last element of *out* being greater than *stop*.

**See also:**

**`numpy.linspace`**

Evenly spaced numbers with careful handling of endpoints.

**`numpy.ogrid`**

Arrays of evenly spaced numbers in N-dimensions.

**`numpy.mgrid`**

Grid-shaped arrays of evenly spaced numbers in N-dimensions.

### Examples

```
>>> np.arange(3)
array([0, 1, 2])
>>> np.arange(3.0)
array([ 0.,  1.,  2.])
>>> np.arange(3,7)
array([3, 4, 5, 6])
>>> np.arange(3,7,2)
array([3, 5])
```

`numpy.`**`linspace`**(*start*, *stop*, *num=50*, *endpoint=True*, *retstep=False*, *dtype=None*, *axis=0*)

Return evenly spaced numbers over a specified interval.

Returns *num* evenly spaced samples, calculated over the interval [*start*, *stop*].

The endpoint of the interval can optionally be excluded.

Changed in version 1.16.0: Non-scalar *start* and *stop* are now supported.

**Parameters**

**start**

[array_like] The starting value of the sequence.

**stop**

[array_like] The end value of the sequence, unless *endpoint* is set to False. In that case, the sequence consists of all but the last of `num + 1` evenly spaced samples, so that *stop* is excluded. Note that the step size changes when *endpoint* is False.

**num**

[int, optional] Number of samples to generate. Default is 50. Must be non-negative.

**endpoint**

[bool, optional] If True, *stop* is the last sample. Otherwise, it is not included. Default is True.

**retstep**

[bool, optional] If True, return (*samples*, *step*), where *step* is the spacing between samples.

**dtype**

[dtype, optional] The type of the output array. If *`dtype`* is not given, infer the data type from the other input arguments.

New in version 1.9.0.

**axis**

[int, optional] The axis in the result to store the samples. Relevant only if start or stop are array-like. By default (0), the samples will be along a new axis inserted at the beginning. Use -1 to get an axis at the end.

New in version 1.16.0.

**Returns**

**samples**

[ndarray] There are *num* equally spaced samples in the closed interval `[start, stop]` or the half-open interval `[start, stop)` (depending on whether *endpoint* is True or False).

**step**

[float, optional] Only returned if *retstep* is True

Size of spacing between samples.

**See also:**

**`arange`**

Similar to `linspace`, but uses a step size (instead of the number of samples).

**`geomspace`**

Similar to `linspace`, but with numbers spaced evenly on a log scale (a geometric progression).

**`logspace`**

Similar to `geomspace`, but with the end points specified as logarithms.

## Examples

```
>>> np.linspace(2.0, 3.0, num=5)
array([2.  , 2.25, 2.5 , 2.75, 3.  ])
>>> np.linspace(2.0, 3.0, num=5, endpoint=False)
array([2. , 2.2, 2.4, 2.6, 2.8])
>>> np.linspace(2.0, 3.0, num=5, retstep=True)
(array([2.  , 2.25, 2.5 , 2.75, 3.  ]), 0.25)
```

Graphical illustration:

```
>>> import matplotlib.pyplot as plt
>>> N = 8
>>> y = np.zeros(N)
>>> x1 = np.linspace(0, 10, N, endpoint=True)
>>> x2 = np.linspace(0, 10, N, endpoint=False)
>>> plt.plot(x1, y, 'o')
[<matplotlib.lines.Line2D object at 0x...>]
>>> plt.plot(x2, y + 0.5, 'o')
[<matplotlib.lines.Line2D object at 0x...>]
>>> plt.ylim([-0.5, 1])
(-0.5, 1)
>>> plt.show()
```

`numpy.`**`logspace`**(*start*, *stop*, *num=50*, *endpoint=True*, *base=10.0*, *dtype=None*, *axis=0*)

Return numbers spaced evenly on a log scale.

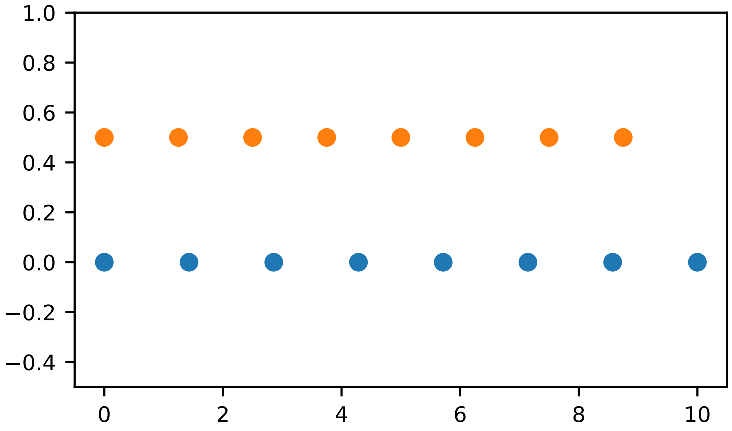
In linear space, the sequence starts at `base ** start` (*base* to the power of *start*) and ends with `base ** stop` (see *endpoint* below).

Changed in version 1.16.0: Non-scalar *start* and *stop* are now supported.

**Parameters**

**start**

[array_like] `base ** start` is the starting value of the sequence.



**stop**

[array_like] `base ** stop` is the final value of the sequence, unless *endpoint* is False. In that case, `num + 1` values are spaced over the interval in log-space, of which all but the last (a sequence of length *num*) are returned.

**num**

[integer, optional] Number of samples to generate. Default is 50.

**endpoint**

[boolean, optional] If true, *stop* is the last sample. Otherwise, it is not included. Default is True.

**base**

[float, optional] The base of the log space. The step size between the elements in `ln(samples) / ln(base)` (or `log_base(samples)`) is uniform. Default is 10.0.

**dtype**

[dtype] The type of the output array. If `dtype` is not given, infer the data type from the other input arguments.

**axis**

[int, optional] The axis in the result to store the samples. Relevant only if start or stop are array-like. By default (0), the samples will be along a new axis inserted at the beginning. Use -1 to get an axis at the end.

New in version 1.16.0.

**Returns**

**samples**

[ndarray] *num* samples, equally spaced on a log scale.

**See also:**

***arange***

Similar to linspace, with the step size specified instead of the number of samples. Note that, when used with a float endpoint, the endpoint may or may not be included.

***linspace***

Similar to logspace, but with the samples uniformly distributed in linear space, instead of log space.

***geomspace***

Similar to logspace, but with endpoints specified directly.

### Notes

Logspace is equivalent to the code

```
>>> y = np.linspace(start, stop, num=num, endpoint=endpoint)
...
>>> power(base, y).astype(dtype)
...
```

### Examples

```
>>> np.logspace(2.0, 3.0, num=4)
array([ 100.        ,  215.443469  ,  464.15888336, 1000.        ])
>>> np.logspace(2.0, 3.0, num=4, endpoint=False)
array([100.        ,  177.827941  ,  316.22776602,  562.34132519])
>>> np.logspace(2.0, 3.0, num=4, base=2.0)
array([4.        ,  5.0396842 ,  6.34960421,  8.        ])
```

Graphical illustration:

```
>>> import matplotlib.pyplot as plt
>>> N = 10
>>> x1 = np.logspace(0.1, 1, N, endpoint=True)
>>> x2 = np.logspace(0.1, 1, N, endpoint=False)
>>> y = np.zeros(N)
>>> plt.plot(x1, y, 'o')
[<matplotlib.lines.Line2D object at 0x...>]
>>> plt.plot(x2, y + 0.5, 'o')
[<matplotlib.lines.Line2D object at 0x...>]
>>> plt.ylim([-0.5, 1])
(-0.5, 1)
>>> plt.show()
```

`numpy.`**`geomspace`**(*start*, *stop*, *num=50*, *endpoint=True*, *dtype=None*, *axis=0*)

Return numbers spaced evenly on a log scale (a geometric progression).

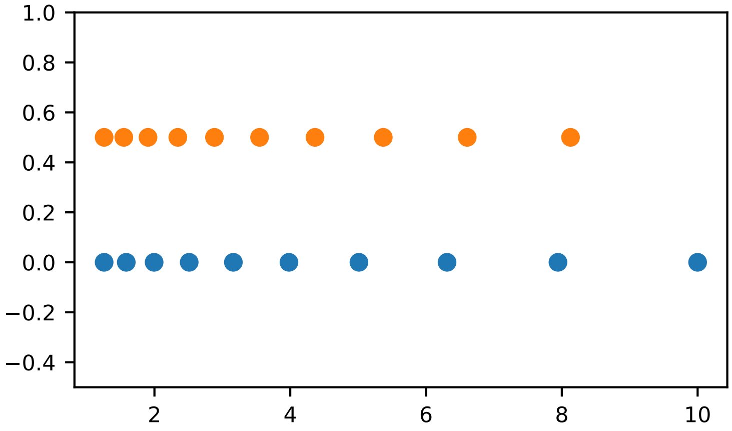
This is similar to *logspace*, but with endpoints specified directly. Each output sample is a constant multiple of the previous.

Changed in version 1.16.0: Non-scalar *start* and *stop* are now supported.

**Parameters**

**start**

[array_like] The starting value of the sequence.



**stop**

[array_like] The final value of the sequence, unless *endpoint* is False. In that case, `num + 1` values are spaced over the interval in log-space, of which all but the last (a sequence of length *num*) are returned.

**num**

[integer, optional] Number of samples to generate. Default is 50.

**endpoint**

[boolean, optional] If true, *stop* is the last sample. Otherwise, it is not included. Default is True.

**dtype**

[dtype] The type of the output array. If `dtype` is not given, infer the data type from the other input arguments.

**axis**

[int, optional] The axis in the result to store the samples. Relevant only if start or stop are array-like. By default (0), the samples will be along a new axis inserted at the beginning. Use -1 to get an axis at the end.

New in version 1.16.0.

**Returns**

**samples**

[ndarray] *num* samples, equally spaced on a log scale.

**See also:**

`logspace`

Similar to geomspace, but with endpoints specified using log and base.

***linspace***

Similar to geomspace, but with arithmetic instead of geometric progression.

***arange***

Similar to linspace, with the step size specified instead of the number of samples.

### Notes

If the inputs or dtype are complex, the output will follow a logarithmic spiral in the complex plane. (There are an infinite number of spirals passing through two points; the output will follow the shortest such path.)

### Examples

```
>>> np.geomspace(1, 1000, num=4)
array([    1.,    10.,   100.,  1000.])
>>> np.geomspace(1, 1000, num=3, endpoint=False)
array([   1.,   10.,  100.])
>>> np.geomspace(1, 1000, num=4, endpoint=False)
array([   1.        ,    5.62341325,   31.6227766 ,  177.827941  ])
>>> np.geomspace(1, 256, num=9)
array([   1.,    2.,    4.,    8.,   16.,   32.,   64.,  128.,  256.])
```

Note that the above may not produce exact integers:

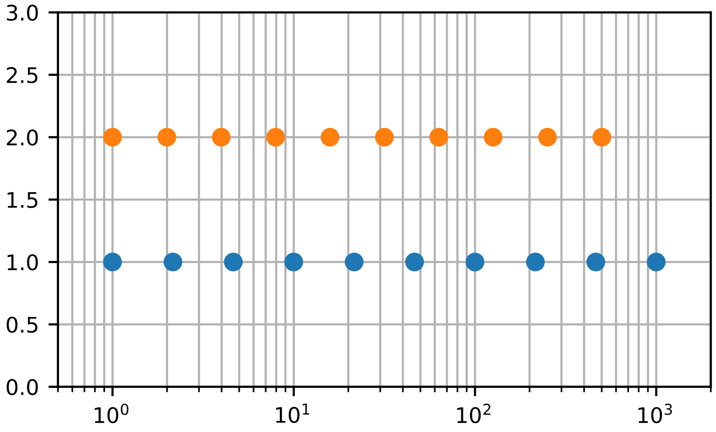
```
>>> np.geomspace(1, 256, num=9, dtype=int)
array([  1,   2,   4,   7,  16,  32,  63, 127, 256])
>>> np.around(np.geomspace(1, 256, num=9)).astype(int)
array([  1,   2,   4,   8,  16,  32,  64, 128, 256])
```

Negative, decreasing, and complex inputs are allowed:

```
>>> np.geomspace(1000, 1, num=4)
array([1000.,  100.,   10.,    1.])
>>> np.geomspace(-1000, -1, num=4)
array([-1000.,  -100.,   -10.,    -1.])
>>> np.geomspace(1j, 1000j, num=4)  # Straight line
array([0.   +1.j, 0.  +10.j, 0. +100.j, 0.+1000.j])
>>> np.geomspace(-1+0j, 1+0j, num=5)  # Circle
array([-1.00000000e+00+1.22464680e-16j, -7.07106781e-01+7.07106781e-01j,
        6.12323400e-17+1.00000000e+00j,  7.07106781e-01+7.07106781e-01j,
        1.00000000e+00+0.00000000e+00j])
```

Graphical illustration of `endpoint` parameter:

```
>>> import matplotlib.pyplot as plt
>>> N = 10
>>> y = np.zeros(N)
>>> plt.semilogx(np.geomspace(1, 1000, N, endpoint=True), y + 1, 'o')
[<matplotlib.lines.Line2D object at 0x...>]
>>> plt.semilogx(np.geomspace(1, 1000, N, endpoint=False), y + 2, 'o')
[<matplotlib.lines.Line2D object at 0x...>]
>>> plt.axis([0.5, 2000, 0, 3])
[0.5, 2000, 0, 3]
>>> plt.grid(True, color='0.7', linestyle='-', which='both', axis='both')
>>> plt.show()
```



`numpy.`**`meshgrid`**(**xi*, *copy=True*, *sparse=False*, *indexing='xy'*)

Return coordinate matrices from coordinate vectors.

Make N-D coordinate arrays for vectorized evaluations of N-D scalar/vector fields over N-D grids, given one-dimensional coordinate arrays x1, x2,…, xn.

Changed in version 1.9: 1-D and 0-D cases are allowed.

**Parameters**

**x1, x2,…, xn**

[array_like] 1-D arrays representing the coordinates of a grid.

**indexing**

[{'xy', 'ij'}, optional] Cartesian ('xy', default) or matrix ('ij') indexing of output. See Notes for more details.

New in version 1.7.0.

**sparse**

[bool, optional] If True a sparse grid is returned in order to conserve memory. Default is False.

New in version 1.7.0.

**copy**

[bool, optional] If False, a view into the original arrays are returned in order to conserve memory. Default is True. Please note that `sparse=False, copy=False` will likely return non-contiguous arrays. Furthermore, more than one element of a broadcast array may refer to a single memory location. If you need to write to the arrays, make copies first.

New in version 1.7.0.

**Returns**

**X1, X2,…, XN**

[ndarray] For vectors *x1*, *x2*,…, 'xn' with lengths `Ni=len(xi)` , return `(N1, N2, N3,...Nn)` shaped arrays if indexing='ij' or `(N2, N1, N3,...Nn)` shaped arrays if indexing='xy' with the elements of *xi* repeated to fill the matrix along the first dimension for *x1*, the second for *x2* and so on.

**See also:**

**`index_tricks.mgrid`**

Construct a multi-dimensional "meshgrid" using indexing notation.

**`index_tricks.ogrid`**

Construct an open multi-dimensional "meshgrid" using indexing notation.

## Notes

This function supports both indexing conventions through the indexing keyword argument. Giving the string 'ij' returns a meshgrid with matrix indexing, while 'xy' returns a meshgrid with Cartesian indexing. In the 2-D case with inputs of length M and N, the outputs are of shape (N, M) for 'xy' indexing and (M, N) for 'ij' indexing. In the 3-D case with inputs of length M, N and P, outputs are of shape (N, M, P) for 'xy' indexing and (M, N, P) for 'ij' indexing. The difference is illustrated by the following code snippet:

```
xv, yv = np.meshgrid(x, y, sparse=False, indexing='ij')
for i in range(nx):
    for j in range(ny):
        # treat xv[i,j], yv[i,j]

xv, yv = np.meshgrid(x, y, sparse=False, indexing='xy')
for i in range(nx):
    for j in range(ny):
        # treat xv[j,i], yv[j,i]
```

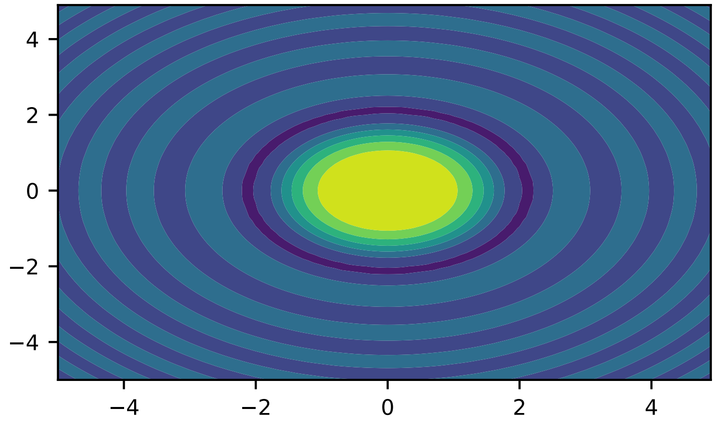
In the 1-D and 0-D case, the indexing and sparse keywords have no effect.

## Examples

```
>>> nx, ny = (3, 2)
>>> x = np.linspace(0, 1, nx)
>>> y = np.linspace(0, 1, ny)
>>> xv, yv = np.meshgrid(x, y)
>>> xv
array([[0. , 0.5, 1. ],
       [0. , 0.5, 1. ]])
>>> yv
array([[0.,  0.,  0.],
       [1.,  1.,  1.]])
>>> xv, yv = np.meshgrid(x, y, sparse=True)  # make sparse output arrays
>>> xv
array([[0. ,  0.5,  1. ]])
>>> yv
array([[0.],
       [1.]])
```

*`meshgrid`* is very useful to evaluate functions on a grid.

```
>>> import matplotlib.pyplot as plt
>>> x = np.arange(-5, 5, 0.1)
>>> y = np.arange(-5, 5, 0.1)
>>> xx, yy = np.meshgrid(x, y, sparse=True)
>>> z = np.sin(xx**2 + yy**2) / (xx**2 + yy**2)
>>> h = plt.contourf(x,y,z)
>>> plt.show()
```



numpy.**mgrid = <numpy.lib.index_tricks.MGridClass object>**

*nd_grid* instance which returns a dense multi-dimensional "meshgrid".

An instance of `numpy.lib.index_tricks.nd_grid` which returns an dense (or fleshed out) mesh-grid when indexed, so that each returned argument has the same shape. The dimensions and number of the output arrays are equal to the number of indexing dimensions. If the step length is not a complex number, then the stop is not inclusive.

However, if the step length is a **complex number** (e.g. 5j), then the integer part of its magnitude is interpreted as specifying the number of points to create between the start and stop values, where the stop value **is inclusive**.

**Returns**

**mesh-grid 'ndarrays' all of the same dimensions**

**See also:**

**`numpy.lib.index_tricks.nd_grid`**

class of *`ogrid`* and *`mgrid`* objects

***`ogrid`***

like mgrid but returns open (not fleshed out) mesh grids

***`r_`***

array concatenator

## Examples

```
>>> np.mgrid[0:5,0:5]
array([[[0, 0, 0, 0, 0],
        [1, 1, 1, 1, 1],
        [2, 2, 2, 2, 2],
        [3, 3, 3, 3, 3],
        [4, 4, 4, 4, 4]],
       [[0, 1, 2, 3, 4],
        [0, 1, 2, 3, 4],
        [0, 1, 2, 3, 4],
        [0, 1, 2, 3, 4],
        [0, 1, 2, 3, 4]]])
>>> np.mgrid[-1:1:5j]
array([-1. , -0.5,  0. ,  0.5,  1. ])
```

`numpy.`**`ogrid = <numpy.lib.index_tricks.OGridClass object>`**

*nd_grid* instance which returns an open multi-dimensional "meshgrid".

An instance of `numpy.lib.index_tricks.nd_grid` which returns an open (i.e. not fleshed out) mesh-grid when indexed, so that only one dimension of each returned array is greater than 1. The dimension and number of the output arrays are equal to the number of indexing dimensions. If the step length is not a complex number, then the stop is not inclusive.

However, if the step length is a **complex number** (e.g. 5j), then the integer part of its magnitude is interpreted as specifying the number of points to create between the start and stop values, where the stop value **is inclusive**.

**Returns**

**mesh-grid**

*ndarrays* with only one dimension not equal to 1

**See also:**

**`np.lib.index_tricks.nd_grid`**

class of `ogrid` and `mgrid` objects

**`mgrid`**

like `ogrid` but returns dense (or fleshed out) mesh grids

**`r_`**

array concatenator

## Examples

```
>>> from numpy import ogrid
>>> ogrid[-1:1:5j]
array([-1. , -0.5,  0. ,  0.5,  1. ])
>>> ogrid[0:5,0:5]
[array([[0],
        [1],
        [2],
        [3],
        [4]]), array([[0, 1, 2, 3, 4]])]
```

### 4.1.6 Building matrices

| | |
|---|---|
| `diag`(v[, k]) | Extract a diagonal or construct a diagonal array. |
| `diagflat`(v[, k]) | Create a two-dimensional array with the flattened input as a diagonal. |
| `tri`(N[, M, k, dtype]) | An array with ones at and below the given diagonal and zeros elsewhere. |
| `tril`(m[, k]) | Lower triangle of an array. |
| `triu`(m[, k]) | Upper triangle of an array. |
| `vander`(x[, N, increasing]) | Generate a Vandermonde matrix. |

`numpy.`**`diag`**(*v*, *k=0*)

Extract a diagonal or construct a diagonal array.

See the more detailed documentation for `numpy.diagonal` if you use this function to extract a diagonal and wish to write to the resulting array; whether it returns a copy or a view depends on what version of numpy you are using.

**Parameters**

**v**

[array_like] If *v* is a 2-D array, return a copy of its *k*-th diagonal. If *v* is a 1-D array, return a 2-D array with *v* on the *k*-th diagonal.

**k**

[int, optional] Diagonal in question. The default is 0. Use *k>0* for diagonals above the main diagonal, and *k<0* for diagonals below the main diagonal.

**Returns**

**out**

[ndarray] The extracted diagonal or constructed diagonal array.

**See also:**

**`diagonal`**

Return specified diagonals.

**`diagflat`**

Create a 2-D array with the flattened input as a diagonal.

**`trace`**

Sum along diagonals.

**`triu`**

Upper triangle of an array.

**`tril`**

Lower triangle of an array.

### Examples

```
>>> x = np.arange(9).reshape((3,3))
>>> x
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
```

```
>>> np.diag(x)
array([0, 4, 8])
>>> np.diag(x, k=1)
array([1, 5])
>>> np.diag(x, k=-1)
array([3, 7])
```

```
>>> np.diag(np.diag(x))
array([[0, 0, 0],
       [0, 4, 0],
       [0, 0, 8]])
```

`numpy.`**`diagflat`**`(`*v*, *k=0*`)`

Create a two-dimensional array with the flattened input as a diagonal.

**Parameters**

**v**

[array_like] Input data, which is flattened and set as the *k*-th diagonal of the output.

**k**

[int, optional] Diagonal to set; 0, the default, corresponds to the “main” diagonal, a positive (negative) *k* giving the number of the diagonal above (below) the main.

**Returns**

**out**

[ndarray] The 2-D output array.

**See also:**

***`diag`***

MATLAB work-alike for 1-D and 2-D arrays.

***`diagonal`***

Return specified diagonals.

***`trace`***

Sum along diagonals.

## Examples

```
>>> np.diagflat([[1,2], [3,4]])
array([[1, 0, 0, 0],
       [0, 2, 0, 0],
       [0, 0, 3, 0],
       [0, 0, 0, 4]])
```

```
>>> np.diagflat([1,2], 1)
array([[0, 1, 0],
       [0, 0, 2],
       [0, 0, 0]])
```

`numpy.`**`tri`**(*N*, *M=None*, *k=0*, *dtype=<class 'float'>*)

An array with ones at and below the given diagonal and zeros elsewhere.

**Parameters**

**N**

[int] Number of rows in the array.

**M**

[int, optional] Number of columns in the array. By default, *M* is taken equal to *N*.

**k**

[int, optional] The sub-diagonal at and below which the array is filled. $k = 0$ is the main diagonal, while $k < 0$ is below it, and $k > 0$ is above. The default is 0.

**dtype**

[dtype, optional] Data type of the returned array. The default is float.

**Returns**

**tri**

[ndarray of shape (N, M)] Array with its lower triangle filled with ones and zero elsewhere; in other words `T[i,j] == 1` for `j <= i + k`, 0 otherwise.

## Examples

```
>>> np.tri(3, 5, 2, dtype=int)
array([[1, 1, 1, 0, 0],
       [1, 1, 1, 1, 0],
       [1, 1, 1, 1, 1]])
```

```
>>> np.tri(3, 5, -1)
array([[0.,  0.,  0.,  0.,  0.],
       [1.,  0.,  0.,  0.,  0.],
       [1.,  1.,  0.,  0.,  0.]])
```

`numpy.`**`tril`**(*m*, *k=0*)

Lower triangle of an array.

Return a copy of an array with elements above the *k*-th diagonal zeroed.

**Parameters**

**m**

[array_like, shape (M, N)] Input array.

**k**

[int, optional] Diagonal above which to zero elements. $k = 0$ (the default) is the main diagonal, $k < 0$ is below it and $k > 0$ is above.

**Returns**

**tril**

[ndarray, shape (M, N)] Lower triangle of *m*, of same shape and data-type as *m*.

**See also:**

**`triu`**

same thing, only for the upper triangle

## Examples

```
>>> np.tril([[1,2,3],[4,5,6],[7,8,9],[10,11,12]], -1)
array([[ 0,  0,  0],
       [ 4,  0,  0],
       [ 7,  8,  0],
       [10, 11, 12]])
```

`numpy.`**`triu`**`(`*m*, *k=0*`)`

Upper triangle of an array.

Return a copy of a matrix with the elements below the *k*-th diagonal zeroed.

Please refer to the documentation for `tril` for further details.

**See also:**

**`tril`**

lower triangle of an array

## Examples

```
>>> np.triu([[1,2,3],[4,5,6],[7,8,9],[10,11,12]], -1)
array([[ 1,  2,  3],
       [ 4,  5,  6],
       [ 0,  8,  9],
       [ 0,  0, 12]])
```

`numpy.`**`vander`**`(`*x*, *N=None*, *increasing=False*`)`

Generate a Vandermonde matrix.

The columns of the output matrix are powers of the input vector. The order of the powers is determined by the *increasing* boolean argument. Specifically, when *increasing* is False, the *i*-th output column is the input vector raised element-wise to the power of `N - i - 1`. Such a matrix with a geometric progression in each row is named for Alexandre- Theophile Vandermonde.

**Parameters**

**x**

[array_like] 1-D input array.

**N**

[int, optional] Number of columns in the output. If *N* is not specified, a square array is returned (`N = len(x)`).

**increasing**

[bool, optional] Order of the powers of the columns. If True, the powers increase from left to right, if False (the default) they are reversed.

New in version 1.9.0.

**Returns**

**out**

[ndarray] Vandermonde matrix. If *increasing* is False, the first column is `x^(N-1)`, the second `x^(N-2)` and so forth. If *increasing* is True, the columns are `x^0, x^1, ..., x^(N-1)`.

**See also:**

*`polynomial.polynomial.polyvander`*

## Examples

```
>>> x = np.array([1, 2, 3, 5])
>>> N = 3
>>> np.vander(x, N)
array([[ 1,  1,  1],
       [ 4,  2,  1],
       [ 9,  3,  1],
       [25,  5,  1]])
```

```
>>> np.column_stack([x**(N-1-i) for i in range(N)])
array([[ 1,  1,  1],
       [ 4,  2,  1],
       [ 9,  3,  1],
       [25,  5,  1]])
```

```
>>> x = np.array([1, 2, 3, 5])
>>> np.vander(x)
array([[  1,   1,   1,   1],
       [  8,   4,   2,   1],
       [ 27,   9,   3,   1],
       [125,  25,   5,   1]])
>>> np.vander(x, increasing=True)
array([[  1,   1,   1,   1],
       [  1,   2,   4,   8],
```

(continues on next page)

(continued from previous page)

```
       [  1,    3,    9,   27],
       [  1,    5,   25,  125]])
```

The determinant of a square Vandermonde matrix is the product of the differences between the values of the input vector:

```
>>> np.linalg.det(np.vander(x))
48.000000000000043 # may vary
>>> (5-3)*(5-2)*(5-1)*(3-2)*(3-1)*(2-1)
48
```

## 4.1.7 The Matrix class

| | |
|---|---|
| *mat*(data[, dtype]) | Interpret the input as a matrix. |
| *bmat*(obj[, ldict, gdict]) | Build a matrix object from a string, nested sequence, or array. |

`numpy.`**`mat`**(*data*, *dtype=None*)

Interpret the input as a matrix.

Unlike *matrix*, *asmatrix* does not make a copy if the input is already a matrix or an ndarray. Equivalent to `matrix(data, copy=False)`.

**Parameters**

**data**

[array_like] Input data.

**dtype**

[data-type] Data-type of the output matrix.

**Returns**

**mat**

[matrix] *data* interpreted as a matrix.

### Examples

```
>>> x = np.array([[1, 2], [3, 4]])
```

```
>>> m = np.asmatrix(x)
```

```
>>> x[0,0] = 5
```

```
>>> m
matrix([[5, 2],
        [3, 4]])
```

## 4.2 Array manipulation routines

### 4.2.1 Basic operations

| | |
|---|---|
| `copyto`(dst, src[, casting, where]) | Copies values from one array to another, broadcasting as necessary. |
| `shape`(a) | Return the shape of an array. |

`numpy.`**`copyto`**(*dst*, *src*, *casting='same_kind'*, *where=True*)

Copies values from one array to another, broadcasting as necessary.

Raises a TypeError if the *casting* rule is violated, and if `where` is provided, it selects which elements to copy.

New in version 1.7.0.

**Parameters**

**dst**

[ndarray] The array into which values are copied.

**src**

[array_like] The array from which values are copied.

**casting**

[{'no', 'equiv', 'safe', 'same_kind', 'unsafe'}, optional] Controls what kind of data casting may occur when copying.

- 'no' means the data types should not be cast at all.
- 'equiv' means only byte-order changes are allowed.
- 'safe' means only casts which can preserve values are allowed.
- 'same_kind' means only safe casts or casts within a kind, like float64 to float32, are allowed.
- 'unsafe' means any data conversions may be done.

**where**

[array_like of bool, optional] A boolean array which is broadcasted to match the dimensions of *dst*, and selects elements to copy from *src* to *dst* wherever it contains the value True.

`numpy.`**`shape`**(*a*)

Return the shape of an array.

**Parameters**

**a**

[array_like] Input array.

**Returns**

**shape**

[tuple of ints] The elements of the shape tuple give the lengths of the corresponding array dimensions.

**See also:**

`alen`

***`ndarray.shape`***

Equivalent array method.

### Examples

```
>>> np.shape(np.eye(3))
(3, 3)
>>> np.shape([[1, 2]])
(1, 2)
>>> np.shape([0])
(1,)
>>> np.shape(0)
()
```

```
>>> a = np.array([(1, 2), (3, 4)], dtype=[('x', 'i4'), ('y', 'i4')])
>>> np.shape(a)
(2,)
>>> a.shape
(2,)
```

## 4.2.2 Changing array shape

| | |
|---|---|
| `reshape`(a, newshape[, order]) | Gives a new shape to an array without changing its data. |
| `ravel`(a[, order]) | Return a contiguous flattened array. |
| `ndarray.flat` | A 1-D iterator over the array. |
| `ndarray.flatten`([order]) | Return a copy of the array collapsed into one dimension. |

`numpy.`**`reshape`**(*a*, *newshape*, *order='C'*)

Gives a new shape to an array without changing its data.

**Parameters**

**a**

[array_like] Array to be reshaped.

**newshape**

[int or tuple of ints] The new shape should be compatible with the original shape. If an integer, then the result will be a 1-D array of that length. One shape dimension can be -1. In this case, the value is inferred from the length of the array and remaining dimensions.

**order**

[{'C', 'F', 'A'}, optional] Read the elements of *a* using this index order, and place the elements into the reshaped array using this index order. 'C' means to read / write the elements using C-like index order, with the last axis index changing fastest, back to the first axis index changing slowest. 'F' means to read / write the elements using Fortran-like index order, with the first index changing fastest, and the last index changing slowest. Note that the 'C' and 'F' options take no account of the memory layout of the underlying array, and only refer to the order of indexing. 'A' means to read / write the elements in Fortran-like index order if *a* is Fortran *contiguous* in memory, C-like order otherwise.

**Returns**

**reshaped_array**

[ndarray] This will be a new view object if possible; otherwise, it will be a copy. Note there is no guarantee of the *memory layout* (C- or Fortran- contiguous) of the returned array.

**See also:**

***ndarray.reshape***

Equivalent method.

## Notes

It is not always possible to change the shape of an array without copying the data. If you want an error to be raised when the data is copied, you should assign the new shape to the shape attribute of the array:

```
>>> a = np.zeros((10, 2))

# A transpose makes the array non-contiguous
>>> b = a.T

# Taking a view makes it possible to modify the shape without modifying
# the initial object.
>>> c = b.view()
>>> c.shape = (20)
Traceback (most recent call last):
   ...
AttributeError: Incompatible shape for in-place modification. Use
`.reshape()` to make a copy with the desired shape.
```

The *order* keyword gives the index ordering both for *fetching* the values from *a*, and then *placing* the values into the output array. For example, let's say you have an array:

```
>>> a = np.arange(6).reshape((3, 2))
>>> a
array([[0, 1],
       [2, 3],
       [4, 5]])
```

You can think of reshaping as first raveling the array (using the given index order), then inserting the elements from the raveled array into the new array using the same kind of index ordering as was used for the raveling.

```
>>> np.reshape(a, (2, 3)) # C-like index ordering
array([[0, 1, 2],
       [3, 4, 5]])
>>> np.reshape(np.ravel(a), (2, 3)) # equivalent to C ravel then C reshape
array([[0, 1, 2],
       [3, 4, 5]])
>>> np.reshape(a, (2, 3), order='F') # Fortran-like index ordering
array([[0, 4, 3],
       [2, 1, 5]])
```

(continues on next page)

(continued from previous page)

```
>>> np.reshape(np.ravel(a, order='F'), (2, 3), order='F')
array([[0, 4, 3],
       [2, 1, 5]])
```

### Examples

```
>>> a = np.array([[1,2,3], [4,5,6]])
>>> np.reshape(a, 6)
array([1, 2, 3, 4, 5, 6])
>>> np.reshape(a, 6, order='F')
array([1, 4, 2, 5, 3, 6])
```

```
>>> np.reshape(a, (3,-1))       # the unspecified value is inferred to be 2
array([[1, 2],
       [3, 4],
       [5, 6]])
```

`numpy.`**`ravel`**(*a*, *order='C'*)

Return a contiguous flattened array.

A 1-D array, containing the elements of the input, is returned. A copy is made only if needed.

As of NumPy 1.10, the returned array will have the same type as the input array. (for example, a masked array will be returned for a masked array input)

**Parameters**

**a**

[array_like] Input array. The elements in *a* are read in the order specified by *order*, and packed as a 1-D array.

**order**

[{'C','F', 'A', 'K'}, optional] The elements of *a* are read using this index order. 'C' means to index the elements in row-major, C-style order, with the last axis index changing fastest, back to the first axis index changing slowest. 'F' means to index the elements in column-major, Fortran-style order, with the first index changing fastest, and the last index changing slowest. Note that the 'C' and 'F' options take no account of the memory layout of the underlying array, and only refer to the order of axis indexing. 'A' means to read the elements in Fortran-like index order if *a* is Fortran *contiguous* in memory, C-like order otherwise. 'K' means to read the elements in the order they occur in memory, except for reversing the data when strides are negative. By default, 'C' index order is used.

**Returns**

**y**

[array_like] y is an array of the same subtype as *a*, with shape `(a.size,)`. Note that matrices are special cased for backward compatibility, if *a* is a matrix, then y is a 1-D ndarray.

**See also:**

***`ndarray.flat`***

1-D iterator over an array.

*`ndarray.flatten`*

1-D array copy of the elements of an array in row-major order.

*`ndarray.reshape`*

Change the shape of an array without changing its data.

## Notes

In row-major, C-style order, in two dimensions, the row index varies the slowest, and the column index the quickest. This can be generalized to multiple dimensions, where row-major order implies that the index along the first axis varies slowest, and the index along the last quickest. The opposite holds for column-major, Fortran-style index ordering.

When a view is desired in as many cases as possible, `arr.reshape(-1)` may be preferable.

## Examples

It is equivalent to `reshape(-1, order=order)`.

```
>>> x = np.array([[1, 2, 3], [4, 5, 6]])
>>> np.ravel(x)
array([1, 2, 3, 4, 5, 6])
```

```
>>> x.reshape(-1)
array([1, 2, 3, 4, 5, 6])
```

```
>>> np.ravel(x, order='F')
array([1, 4, 2, 5, 3, 6])
```

When `order` is ‘A’, it will preserve the array’s ‘C’ or ‘F’ ordering:

```
>>> np.ravel(x.T)
array([1, 4, 2, 5, 3, 6])
>>> np.ravel(x.T, order='A')
array([1, 2, 3, 4, 5, 6])
```

When `order` is ‘K’, it will preserve orderings that are neither ‘C’ nor ‘F’, but won’t reverse axes:

```
>>> a = np.arange(3)[::-1]; a
array([2, 1, 0])
>>> a.ravel(order='C')
array([2, 1, 0])
>>> a.ravel(order='K')
array([2, 1, 0])
```

```
>>> a = np.arange(12).reshape(2,3,2).swapaxes(1,2); a
array([[[ 0,  2,  4],
        [ 1,  3,  5]],
       [[ 6,  8, 10],
        [ 7,  9, 11]]])
>>> a.ravel(order='C')
array([ 0,  2,  4,  1,  3,  5,  6,  8, 10,  7,  9, 11])
>>> a.ravel(order='K')
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])
```

### 4.2.3 Transpose-like operations

| | |
|---|---|
| *moveaxis*(a, source, destination) | Move axes of an array to new positions. |
| *rollaxis*(a, axis[, start]) | Roll the specified axis backwards, until it lies in a given position. |
| *swapaxes*(a, axis1, axis2) | Interchange two axes of an array. |
| *ndarray.T* | The transposed array. |
| *transpose*(a[, axes]) | Reverse or permute the axes of an array; returns the modified array. |

numpy.**moveaxis**(*a*, *source*, *destination*)

Move axes of an array to new positions.

Other axes remain in their original order.

New in version 1.11.0.

**Parameters**

**a**

[np.ndarray] The array whose axes should be reordered.

**source**

[int or sequence of int] Original positions of the axes to move. These must be unique.

**destination**

[int or sequence of int] Destination positions for each of the original axes. These must also be unique.

**Returns**

**result**

[np.ndarray] Array with moved axes. This array is a view of the input array.

**See also:**

***transpose***

Permute the dimensions of an array.

***swapaxes***

Interchange two axes of an array.

#### Examples

```
>>> x = np.zeros((3, 4, 5))
>>> np.moveaxis(x, 0, -1).shape
(4, 5, 3)
>>> np.moveaxis(x, -1, 0).shape
(5, 3, 4)
```

These all achieve the same result:

```
>>> np.transpose(x).shape
(5, 4, 3)
>>> np.swapaxes(x, 0, -1).shape
(5, 4, 3)
>>> np.moveaxis(x, [0, 1], [-1, -2]).shape
(5, 4, 3)
>>> np.moveaxis(x, [0, 1, 2], [-1, -2, -3]).shape
(5, 4, 3)
```

`numpy.`**`rollaxis`**(*a*, *axis*, *start=0*)

Roll the specified axis backwards, until it lies in a given position.

This function continues to be supported for backward compatibility, but you should prefer *`moveaxis`*. The *`moveaxis`* function was added in NumPy 1.11.

**Parameters**

**a**

[ndarray] Input array.

**axis**

[int] The axis to be rolled. The positions of the other axes do not change relative to one another.

**start**

[int, optional] When `start <= axis`, the axis is rolled back until it lies in this position. When `start > axis`, the axis is rolled until it lies before this position. The default, 0, results in a "complete" roll. The following table describes how negative values of `start` are interpreted:

| `start` | Normalized `start` |
|---|---|
| `-(arr.ndim+1)` | raise `AxisError` |
| `-arr.ndim` | 0 |
| ⋮ | ⋮ |
| `-1` | `arr.ndim-1` |
| `0` | `0` |
| ⋮ | ⋮ |
| `arr.ndim` | `arr.ndim` |
| `arr.ndim + 1` | raise `AxisError` |

**Returns**

**res**

[ndarray] For NumPy >= 1.10.0 a view of *a* is always returned. For earlier NumPy versions a view of *a* is returned only if the order of the axes is changed, otherwise the input array is returned.

**See also:**

***`moveaxis`***

Move array axes to new positions.

***`roll`***

Roll the elements of an array by a number of positions along a given axis.

### Examples

```
>>> a = np.ones((3,4,5,6))
>>> np.rollaxis(a, 3, 1).shape
(3, 6, 4, 5)
>>> np.rollaxis(a, 2).shape
(5, 3, 4, 6)
>>> np.rollaxis(a, 1, 4).shape
(3, 5, 6, 4)
```

`numpy.`**`swapaxes`**`(`*a*, *axis1*, *axis2*`)`

Interchange two axes of an array.

**Parameters**

**a**

[array_like] Input array.

**axis1**

[int] First axis.

**axis2**

[int] Second axis.

**Returns**

**a_swapped**

[ndarray] For NumPy >= 1.10.0, if *a* is an ndarray, then a view of *a* is returned; otherwise a new array is created. For earlier NumPy versions a view of *a* is returned only if the order of the axes is changed, otherwise the input array is returned.

### Examples

```
>>> x = np.array([[1,2,3]])
>>> np.swapaxes(x,0,1)
array([[1],
       [2],
       [3]])
```

```
>>> x = np.array([[[0,1],[2,3]],[[4,5],[6,7]]])
>>> x
array([[[0, 1],
        [2, 3]],
       [[4, 5],
        [6, 7]]])
```

```
>>> np.swapaxes(x,0,2)
array([[[0, 4],
        [2, 6]],
       [[1, 5],
        [3, 7]]])
```

`numpy.transpose`(*a*, *axes=None*)

Reverse or permute the axes of an array; returns the modified array.

For an array a with two axes, transpose(a) gives the matrix transpose.

**Parameters**

**a**

[array_like] Input array.

**axes**

[tuple or list of ints, optional] If specified, it must be a tuple or list which contains a permutation of [0,1,..,N-1] where N is the number of axes of a. The i'th axis of the returned array will correspond to the axis numbered `axes[i]` of the input. If not specified, defaults to `range(a.ndim)[::-1]`, which reverses the order of the axes.

**Returns**

**p**

[ndarray] *a* with its axes permuted. A view is returned whenever possible.

**See also:**

*moveaxis*, *argsort*

## Notes

Use *transpose(a, argsort(axes))* to invert the transposition of tensors when using the *axes* keyword argument.

Transposing a 1-D array returns an unchanged view of the original array.

## Examples

```
>>> x = np.arange(4).reshape((2,2))
>>> x
array([[0, 1],
       [2, 3]])
```

```
>>> np.transpose(x)
array([[0, 2],
       [1, 3]])
```

```
>>> x = np.ones((1, 2, 3))
>>> np.transpose(x, (1, 0, 2)).shape
(2, 1, 3)
```

### 4.2.4 Changing number of dimensions

| | |
|---|---|
| *atleast_1d*(*arys) | Convert inputs to arrays with at least one dimension. |
| *atleast_2d*(*arys) | View inputs as arrays with at least two dimensions. |
| *atleast_3d*(*arys) | View inputs as arrays with at least three dimensions. |
| *broadcast* | Produce an object that mimics broadcasting. |
| *broadcast_to*(array, shape[, subok]) | Broadcast an array to a new shape. |
| *broadcast_arrays*(*args[, subok]) | Broadcast any number of arrays against each other. |
| *expand_dims*(a, axis) | Expand the shape of an array. |
| *squeeze*(a[, axis]) | Remove single-dimensional entries from the shape of an array. |

`numpy.`**`atleast_1d`**`(`*`*arys`*`)`

Convert inputs to arrays with at least one dimension.

Scalar inputs are converted to 1-dimensional arrays, whilst higher-dimensional inputs are preserved.

**Parameters**

**arys1, arys2, …**

[array_like] One or more input arrays.

**Returns**

**ret**

[ndarray] An array, or list of arrays, each with `a.ndim >= 1`. Copies are made only if necessary.

**See also:**

*atleast_2d*, *atleast_3d*

#### Examples

```
>>> np.atleast_1d(1.0)
array([1.])
```

```
>>> x = np.arange(9.0).reshape(3,3)
>>> np.atleast_1d(x)
array([[0., 1., 2.],
       [3., 4., 5.],
       [6., 7., 8.]])
>>> np.atleast_1d(x) is x
True
```

```
>>> np.atleast_1d(1, [3, 4])
[array([1]), array([3, 4])]
```

`numpy.`**`atleast_2d`**`(`*`*arys`*`)`

View inputs as arrays with at least two dimensions.

**Parameters**

**arys1, arys2, …**

[array_like] One or more array-like sequences. Non-array inputs are converted to arrays. Arrays that already have two or more dimensions are preserved.

**Returns**

**res, res2, …**

[ndarray] An array, or list of arrays, each with `a.ndim >= 2`. Copies are avoided where possible, and views with two or more dimensions are returned.

**See also:**

*atleast_1d*, *atleast_3d*

## Examples

```
>>> np.atleast_2d(3.0)
array([[3.]])
```

```
>>> x = np.arange(3.0)
>>> np.atleast_2d(x)
array([[0., 1., 2.]])
>>> np.atleast_2d(x).base is x
True
```

```
>>> np.atleast_2d(1, [1, 2], [[1, 2]])
[array([[1]]), array([[1, 2]]), array([[1, 2]])]
```

`numpy.`**`atleast_3d`**`(`**arys*`)`

View inputs as arrays with at least three dimensions.

**Parameters**

**arys1, arys2, …**

[array_like] One or more array-like sequences. Non-array inputs are converted to arrays. Arrays that already have three or more dimensions are preserved.

**Returns**

**res1, res2, …**

[ndarray] An array, or list of arrays, each with `a.ndim >= 3`. Copies are avoided where possible, and views with three or more dimensions are returned. For example, a 1-D array of shape `(N,)` becomes a view of shape `(1, N, 1)`, and a 2-D array of shape `(M, N)` becomes a view of shape `(M, N, 1)`.

**See also:**

*atleast_1d*, *atleast_2d*

### Examples

```
>>> np.atleast_3d(3.0)
array([[[3.]]])
```

```
>>> x = np.arange(3.0)
>>> np.atleast_3d(x).shape
(1, 3, 1)
```

```
>>> x = np.arange(12.0).reshape(4,3)
>>> np.atleast_3d(x).shape
(4, 3, 1)
>>> np.atleast_3d(x).base is x.base  # x is a reshape, so not base itself
True
```

```
>>> for arr in np.atleast_3d([1, 2], [[1, 2]], [[[1, 2]]]):
...     print(arr, arr.shape)
...
[[[1]
  [2]]] (1, 2, 1)
[[[1]
  [2]]] (1, 2, 1)
[[[1 2]]] (1, 1, 2)
```

`numpy.`**`broadcast_to`**(*array*, *shape*, *subok=False*)

Broadcast an array to a new shape.

**Parameters**

**array**

[array_like] The array to broadcast.

**shape**

[tuple] The shape of the desired array.

**subok**

[bool, optional] If True, then sub-classes will be passed-through, otherwise the returned array will be forced to be a base-class array (default).

**Returns**

**broadcast**

[array] A readonly view on the original array with the given shape. It is typically not contiguous. Furthermore, more than one element of a broadcasted array may refer to a single memory location.

**Raises**

**ValueError**

If the array is not compatible with the new shape according to NumPy's broadcasting rules.

### Notes

New in version 1.10.0.

### Examples

```
>>> x = np.array([1, 2, 3])
>>> np.broadcast_to(x, (3, 3))
array([[1, 2, 3],
       [1, 2, 3],
       [1, 2, 3]])
```

`numpy.`**`broadcast_arrays`**(**args*, *subok=False*)

Broadcast any number of arrays against each other.

**Parameters**

**‘*args‘**

[array_likes] The arrays to broadcast.

**subok**

[bool, optional] If True, then sub-classes will be passed-through, otherwise the returned arrays will be forced to be a base-class array (default).

**Returns**

**broadcasted**

[list of arrays] These arrays are views on the original arrays. They are typically not contiguous. Furthermore, more than one element of a broadcasted array may refer to a single memory location. If you need to write to the arrays, make copies first. While you can set the `writable` flag True, writing to a single output value may end up changing more than one location in the output array.

Deprecated since version 1.17: The output is currently marked so that if written to, a deprecation warning will be emitted. A future version will set the `writable` flag False so writing to it will raise an error.

### Examples

```
>>> x = np.array([[1,2,3]])
>>> y = np.array([[4],[5]])
>>> np.broadcast_arrays(x, y)
[array([[1, 2, 3],
       [1, 2, 3]]), array([[4, 4, 4],
       [5, 5, 5]])]
```

Here is a useful idiom for getting contiguous copies instead of non-contiguous views.

```
>>> [np.array(a) for a in np.broadcast_arrays(x, y)]
[array([[1, 2, 3],
       [1, 2, 3]]), array([[4, 4, 4],
       [5, 5, 5]])]
```

`numpy.`**`expand_dims`**(*a*, *axis*)

Expand the shape of an array.

Insert a new axis that will appear at the *axis* position in the expanded array shape.

**Parameters**

**a**

[array_like] Input array.

**axis**

[int or tuple of ints] Position in the expanded axes where the new axis (or axes) is placed.

Deprecated since version 1.13.0: Passing an axis where `axis > a.ndim` will be treated as `axis == a.ndim`, and passing `axis < -a.ndim - 1` will be treated as `axis == 0`. This behavior is deprecated.

Changed in version 1.18.0: A tuple of axes is now supported. Out of range axes as described above are now forbidden and raise an `AxisError`.

**Returns**

**result**

[ndarray] View of *a* with the number of dimensions increased.

**See also:**

***`squeeze`***

The inverse operation, removing singleton dimensions

***`reshape`***

Insert, remove, and combine dimensions, and resize existing ones

`doc.indexing`, *`atleast_1d`*, *`atleast_2d`*, *`atleast_3d`*

## Examples

```
>>> x = np.array([1, 2])
>>> x.shape
(2,)
```

The following is equivalent to `x[np.newaxis, :]` or `x[np.newaxis]`:

```
>>> y = np.expand_dims(x, axis=0)
>>> y
array([[1, 2]])
>>> y.shape
(1, 2)
```

The following is equivalent to `x[:, np.newaxis]`:

```
>>> y = np.expand_dims(x, axis=1)
>>> y
array([[1],
       [2]])
>>> y.shape
(2, 1)
```

`axis` may also be a tuple:

```
>>> y = np.expand_dims(x, axis=(0, 1))
>>> y
array([[[1, 2]]])
```

```
>>> y = np.expand_dims(x, axis=(2, 0))
>>> y
array([[[1],
        [2]]])
```

Note that some examples may use `None` instead of `np.newaxis`. These are the same objects:

```
>>> np.newaxis is None
True
```

`numpy.`**`squeeze`**(*a*, *axis=None*)

Remove single-dimensional entries from the shape of an array.

**Parameters**

**a**

[array_like] Input data.

**axis**

[None or int or tuple of ints, optional] New in version 1.7.0.

Selects a subset of the single-dimensional entries in the shape. If an axis is selected with shape entry greater than one, an error is raised.

**Returns**

**squeezed**

[ndarray] The input array, but with all or a subset of the dimensions of length 1 removed. This is always *a* itself or a view into *a*. Note that if all axes are squeezed, the result is a 0d array and not a scalar.

**Raises**

**ValueError**

If *axis* is not None, and an axis being squeezed is not of length 1

**See also:**

*`expand_dims`*

The inverse operation, adding singleton dimensions

***reshape***

Insert, remove, and combine dimensions, and resize existing ones

### Examples

```
>>> x = np.array([[[0], [1], [2]]])
>>> x.shape
(1, 3, 1)
>>> np.squeeze(x).shape
(3,)
>>> np.squeeze(x, axis=0).shape
(3, 1)
>>> np.squeeze(x, axis=1).shape
Traceback (most recent call last):
...
ValueError: cannot select an axis to squeeze out which has size not equal to one
>>> np.squeeze(x, axis=2).shape
(1, 3)
>>> x = np.array([[1234]])
>>> x.shape
(1, 1)
>>> np.squeeze(x)
array(1234)  # 0d array
>>> np.squeeze(x).shape
()
>>> np.squeeze(x)[()]
1234
```

## 4.2.5 Changing kind of array

| | |
|---|---|
| *asarray*(a[, dtype, order]) | Convert the input to an array. |
| *asanyarray*(a[, dtype, order]) | Convert the input to an ndarray, but pass ndarray subclasses through. |
| *asmatrix*(data[, dtype]) | Interpret the input as a matrix. |
| *asfarray*(a[, dtype]) | Return an array converted to a float type. |
| *asfortranarray*(a[, dtype]) | Return an array (ndim >= 1) laid out in Fortran order in memory. |
| *ascontiguousarray*(a[, dtype]) | Return a contiguous array (ndim >= 1) in memory (C order). |
| *asarray_chkfinite*(a[, dtype, order]) | Convert the input to an array, checking for NaNs or Infs. |
| *asscalar*(a) | Convert an array of size 1 to its scalar equivalent. |
| *require*(a[, dtype, requirements]) | Return an ndarray of the provided type that satisfies requirements. |

numpy.**asfarray**(*a*, *dtype=<class 'numpy.float64'>*)

Return an array converted to a float type.

**Parameters**

**a**

[array_like] The input array.

**dtype**

[str or dtype object, optional] Float type code to coerce input array *a*. If `dtype` is one of the 'int' dtypes, it is replaced with float64.

**Returns**

**out**

[ndarray] The input *a* as a float ndarray.

## Examples

```
>>> np.asfarray([2, 3])
array([2.,  3.])
>>> np.asfarray([2, 3], dtype='float')
array([2.,  3.])
>>> np.asfarray([2, 3], dtype='int8')
array([2.,  3.])
```

`numpy.`**`asfortranarray`**(*a*, *dtype=None*)

Return an array (ndim >= 1) laid out in Fortran order in memory.

**Parameters**

**a**

[array_like] Input array.

**dtype**

[str or dtype object, optional] By default, the data-type is inferred from the input data.

**Returns**

**out**

[ndarray] The input *a* in Fortran, or column-major, order.

**See also:**

***`ascontiguousarray`***

Convert input to a contiguous (C order) array.

***`asanyarray`***

Convert input to an ndarray with either row or column-major memory order.

***`require`***

Return an ndarray that satisfies requirements.

***`ndarray.flags`***

Information about the memory layout of the array.

### Examples

```
>>> x = np.arange(6).reshape(2,3)
>>> y = np.asfortranarray(x)
>>> x.flags['F_CONTIGUOUS']
False
>>> y.flags['F_CONTIGUOUS']
True
```

Note: This function returns an array with at least one-dimension (1-d) so it will not preserve 0-d arrays.

`numpy.`**`asarray_chkfinite`**(*a*, *dtype=None*, *order=None*)

Convert the input to an array, checking for NaNs or Infs.

**Parameters**

**a**

[array_like] Input data, in any form that can be converted to an array. This includes lists, lists of tuples, tuples, tuples of tuples, tuples of lists and ndarrays. Success requires no NaNs or Infs.

**dtype**

[data-type, optional] By default, the data-type is inferred from the input data.

**order**

[{'C', 'F'}, optional] Whether to use row-major (C-style) or column-major (Fortran-style) memory representation. Defaults to 'C'.

**Returns**

**out**

[ndarray] Array interpretation of *a*. No copy is performed if the input is already an ndarray. If *a* is a subclass of ndarray, a base class ndarray is returned.

**Raises**

**ValueError**

Raises ValueError if *a* contains NaN (Not a Number) or Inf (Infinity).

**See also:**

***`asarray`***

Create and array.

***`asanyarray`***

Similar function which passes through subclasses.

***`ascontiguousarray`***

Convert input to a contiguous array.

***`asfarray`***

Convert input to a floating point ndarray.

***asfortranarray***

Convert input to an ndarray with column-major memory order.

***fromiter***

Create an array from an iterator.

***fromfunction***

Construct an array by executing a function on grid positions.

### Examples

Convert a list into an array. If all elements are finite `asarray_chkfinite` is identical to `asarray`.

```
>>> a = [1, 2]
>>> np.asarray_chkfinite(a, dtype=float)
array([1., 2.])
```

Raises ValueError if array_like contains Nans or Infs.

```
>>> a = [1, 2, np.inf]
>>> try:
...     np.asarray_chkfinite(a)
... except ValueError:
...     print('ValueError')
...
ValueError
```

`numpy.`**`asscalar`**`(`*a*`)`

Convert an array of size 1 to its scalar equivalent.

Deprecated since version 1.16: Deprecated, use *numpy.ndarray.item()* instead.

**Parameters**

**a**

[ndarray] Input array of size 1.

**Returns**

**out**

[scalar] Scalar representation of *a*. The output data type is the same type returned by the input's *item* method.

### Examples

```
>>> np.asscalar(np.array([24]))
24
```

`numpy.`**`require`**`(`*a*, *dtype=None*, *requirements=None*`)`

Return an ndarray of the provided type that satisfies requirements.

This function is useful to be sure that an array with the correct flags is returned for passing to compiled code (perhaps through ctypes).

**Parameters**

**a**

[array_like] The object to be converted to a type-and-requirement-satisfying array.

**dtype**

[data-type] The required data-type. If None preserve the current dtype. If your application requires the data to be in native byteorder, include a byteorder specification as a part of the dtype specification.

**requirements**

[str or list of str] The requirements list can be any of the following

- 'F_CONTIGUOUS' ('F') - ensure a Fortran-contiguous array
- 'C_CONTIGUOUS' ('C') - ensure a C-contiguous array
- 'ALIGNED' ('A') - ensure a data-type aligned array
- 'WRITEABLE' ('W') - ensure a writable array
- 'OWNDATA' ('O') - ensure an array that owns its own data
- 'ENSUREARRAY', ('E') - ensure a base array, instead of a subclass

**Returns**

**out**

[ndarray] Array with specified requirements and type if given.

**See also:**

`asarray`

Convert input to an ndarray.

`asanyarray`

Convert to an ndarray, but pass through ndarray subclasses.

`ascontiguousarray`

Convert input to a contiguous array.

`asfortranarray`

Convert input to an ndarray with column-major memory order.

`ndarray.flags`

Information about the memory layout of the array.

### Notes

The returned array will be guaranteed to have the listed requirements by making a copy if needed.

### Examples

```
>>> x = np.arange(6).reshape(2,3)
>>> x.flags
  C_CONTIGUOUS : True
  F_CONTIGUOUS : False
  OWNDATA : False
  WRITEABLE : True
  ALIGNED : True
  WRITEBACKIFCOPY : False
  UPDATEIFCOPY : False
```

```
>>> y = np.require(x, dtype=np.float32, requirements=['A', 'O', 'W', 'F'])
>>> y.flags
  C_CONTIGUOUS : False
  F_CONTIGUOUS : True
  OWNDATA : True
  WRITEABLE : True
  ALIGNED : True
  WRITEBACKIFCOPY : False
  UPDATEIFCOPY : False
```

## 4.2.6 Joining arrays

| | |
|---|---|
| `concatenate`([axis, out]) | Join a sequence of arrays along an existing axis. |
| `stack`(arrays[, axis, out]) | Join a sequence of arrays along a new axis. |
| `block`(arrays) | Assemble an nd-array from nested lists of blocks. |
| `vstack`(tup) | Stack arrays in sequence vertically (row wise). |
| `hstack`(tup) | Stack arrays in sequence horizontally (column wise). |
| `dstack`(tup) | Stack arrays in sequence depth wise (along third axis). |
| `column_stack`(tup) | Stack 1-D arrays as columns into a 2-D array. |

`numpy.`**`concatenate`**(*(a1, a2, ...), axis=0, out=None*)

Join a sequence of arrays along an existing axis.

**Parameters**

**a1, a2, ...**

[sequence of array_like] The arrays must have the same shape, except in the dimension corresponding to *axis* (the first, by default).

**axis**

[int, optional] The axis along which the arrays will be joined. If axis is None, arrays are flattened before use. Default is 0.

**out**

[ndarray, optional] If provided, the destination to place the result. The shape must be correct, matching that of what concatenate would have returned if no out argument were specified.

**Returns**

**res**

[ndarray] The concatenated array.

**See also:**

**`ma.concatenate`**

Concatenate function that preserves input masks.

**`array_split`**

Split an array into multiple sub-arrays of equal or near-equal size.

**`split`**

Split array into a list of multiple sub-arrays of equal size.

**`hsplit`**

Split array into multiple sub-arrays horizontally (column wise).

**`vsplit`**

Split array into multiple sub-arrays vertically (row wise).

**`dsplit`**

Split array into multiple sub-arrays along the 3rd axis (depth).

**`stack`**

Stack a sequence of arrays along a new axis.

**`block`**

Assemble arrays from blocks.

**`hstack`**

Stack arrays in sequence horizontally (column wise).

**`vstack`**

Stack arrays in sequence vertically (row wise).

**`dstack`**

Stack arrays in sequence depth wise (along third dimension).

**`column_stack`**

Stack 1-D arrays as columns into a 2-D array.

### Notes

When one or more of the arrays to be concatenated is a MaskedArray, this function will return a MaskedArray object instead of an ndarray, but the input masks are *not* preserved. In cases where a MaskedArray is expected as input, use the ma.concatenate function from the masked array module instead.

### Examples

```
>>> a = np.array([[1, 2], [3, 4]])
>>> b = np.array([[5, 6]])
>>> np.concatenate((a, b), axis=0)
array([[1, 2],
       [3, 4],
       [5, 6]])
>>> np.concatenate((a, b.T), axis=1)
array([[1, 2, 5],
       [3, 4, 6]])
>>> np.concatenate((a, b), axis=None)
array([1, 2, 3, 4, 5, 6])
```

This function will not preserve masking of MaskedArray inputs.

```
>>> a = np.ma.arange(3)
>>> a[1] = np.ma.masked
>>> b = np.arange(2, 5)
>>> a
masked_array(data=[0, --, 2],
             mask=[False,  True, False],
       fill_value=999999)
>>> b
array([2, 3, 4])
>>> np.concatenate([a, b])
masked_array(data=[0, 1, 2, 2, 3, 4],
             mask=False,
       fill_value=999999)
>>> np.ma.concatenate([a, b])
masked_array(data=[0, --, 2, 2, 3, 4],
             mask=[False,  True, False, False, False, False],
       fill_value=999999)
```

`numpy.`**`stack`**(*arrays*, *axis=0*, *out=None*)

Join a sequence of arrays along a new axis.

The `axis` parameter specifies the index of the new axis in the dimensions of the result. For example, if `axis=0` it will be the first dimension and if `axis=-1` it will be the last dimension.

New in version 1.10.0.

**Parameters**

**arrays**

[sequence of array_like] Each array must have the same shape.

**axis**

[int, optional] The axis in the result array along which the input arrays are stacked.

**out**

[ndarray, optional] If provided, the destination to place the result. The shape must be correct, matching that of what stack would have returned if no out argument were specified.

**Returns**

**stacked**

[ndarray] The stacked array has one more dimension than the input arrays.

**See also:**

***`concatenate`***

Join a sequence of arrays along an existing axis.

***`block`***

Assemble an nd-array from nested lists of blocks.

***`split`***

Split array into a list of multiple sub-arrays of equal size.

### Examples

```
>>> arrays = [np.random.randn(3, 4) for _ in range(10)]
>>> np.stack(arrays, axis=0).shape
(10, 3, 4)
```

```
>>> np.stack(arrays, axis=1).shape
(3, 10, 4)
```

```
>>> np.stack(arrays, axis=2).shape
(3, 4, 10)
```

```
>>> a = np.array([1, 2, 3])
>>> b = np.array([2, 3, 4])
>>> np.stack((a, b))
array([[1, 2, 3],
       [2, 3, 4]])
```

```
>>> np.stack((a, b), axis=-1)
array([[1, 2],
       [2, 3],
       [3, 4]])
```

`numpy.`**`block`**(*arrays*)

Assemble an nd-array from nested lists of blocks.

Blocks in the innermost lists are concatenated (see *`concatenate`*) along the last dimension (-1), then these are concatenated along the second-last dimension (-2), and so on until the outermost list is reached.

Blocks can be of any dimension, but will not be broadcasted using the normal rules. Instead, leading axes of size 1 are inserted, to make `block.ndim` the same for all blocks. This is primarily useful for working with scalars, and means that code like `np.block([v, 1])` is valid, where `v.ndim == 1`.

When the nested list is two levels deep, this allows block matrices to be constructed from their components.

New in version 1.13.0.

**Parameters**

**arrays**

[nested list of array_like or scalars (but not tuples)] If passed a single ndarray or scalar (a nested list of depth 0), this is returned unmodified (and not copied).

Elements shapes must match along the appropriate axes (without broadcasting), but leading 1s will be prepended to the shape as necessary to make the dimensions match.

**Returns**

**block_array**

[ndarray] The array assembled from the given blocks.

The dimensionality of the output is equal to the greatest of: * the dimensionality of all the inputs * the depth to which the input list is nested

**Raises**

**ValueError**

- If list depths are mismatched - for instance, `[[a, b], c]` is illegal, and should be spelt `[[a, b], [c]]`
- If lists are empty - for instance, `[[a, b], []]`

**See also:**

**`concatenate`**

Join a sequence of arrays along an existing axis.

**`stack`**

Join a sequence of arrays along a new axis.

**`vstack`**

Stack arrays in sequence vertically (row wise).

**`hstack`**

Stack arrays in sequence horizontally (column wise).

**`dstack`**

Stack arrays in sequence depth wise (along third axis).

**`column_stack`**

Stack 1-D arrays as columns into a 2-D array.

**`vsplit`**

Split an array into multiple sub-arrays vertically (row-wise).

## Notes

When called with only scalars, `np.block` is equivalent to an ndarray call. So `np.block([[1, 2], [3, 4]])` is equivalent to `np.array([[1, 2], [3, 4]])`.

This function does not enforce that the blocks lie on a fixed grid. `np.block([[a, b], [c, d]])` is not restricted to arrays of the form:

```
AAAbb
AAAbb
cccDD
```

But is also allowed to produce, for some `a, b, c, d`:

```
AAAbb
AAAbb
cDDDD
```

Since concatenation happens along the last axis first, *`block`* is _not_ capable of producing the following directly:

```
AAAbb
cccbb
cccDD
```

Matlab's "square bracket stacking", `[A, B, ...; p, q, ...]`, is equivalent to `np.block([[A, B, ...], [p, q, ...]])`.

## Examples

The most common use of this function is to build a block matrix

```
>>> A = np.eye(2) * 2
>>> B = np.eye(3) * 3
>>> np.block([
...     [A,               np.zeros((2, 3))],
...     [np.ones((3, 2)), B               ]
... ])
array([[2., 0., 0., 0., 0.],
       [0., 2., 0., 0., 0.],
       [1., 1., 3., 0., 0.],
       [1., 1., 0., 3., 0.],
       [1., 1., 0., 0., 3.]])
```

With a list of depth 1, *`block`* can be used as *`hstack`*

```
>>> np.block([1, 2, 3])              # hstack([1, 2, 3])
array([1, 2, 3])
```

```
>>> a = np.array([1, 2, 3])
>>> b = np.array([2, 3, 4])
>>> np.block([a, b, 10])             # hstack([a, b, 10])
array([ 1,  2,  3,  2,  3,  4, 10])
```

```
>>> A = np.ones((2, 2), int)
>>> B = 2 * A
>>> np.block([A, B])                 # hstack([A, B])
```

(continues on next page)

(continued from previous page)

```
array([[1, 1, 2, 2],
       [1, 1, 2, 2]])
```

With a list of depth 2, `block` can be used in place of `vstack`:

```
>>> a = np.array([1, 2, 3])
>>> b = np.array([2, 3, 4])
>>> np.block([[a], [b]])             # vstack([a, b])
array([[1, 2, 3],
       [2, 3, 4]])
```

```
>>> A = np.ones((2, 2), int)
>>> B = 2 * A
>>> np.block([[A], [B]])             # vstack([A, B])
array([[1, 1],
       [1, 1],
       [2, 2],
       [2, 2]])
```

It can also be used in places of `atleast_1d` and `atleast_2d`

```
>>> a = np.array(0)
>>> b = np.array([1])
>>> np.block([a])                    # atleast_1d(a)
array([0])
>>> np.block([b])                    # atleast_1d(b)
array([1])
```

```
>>> np.block([[a]])                  # atleast_2d(a)
array([[0]])
>>> np.block([[b]])                  # atleast_2d(b)
array([[1]])
```

`numpy.`**`vstack`**`(`*tup*`)`

Stack arrays in sequence vertically (row wise).

This is equivalent to concatenation along the first axis after 1-D arrays of shape *(N,)* have been reshaped to *(1,N)*. Rebuilds arrays divided by `vsplit`.

This function makes most sense for arrays with up to 3 dimensions. For instance, for pixel-data with a height (first axis), width (second axis), and r/g/b channels (third axis). The functions `concatenate`, `stack` and `block` provide more general stacking and concatenation operations.

**Parameters**

**tup**

[sequence of ndarrays] The arrays must have the same shape along all but the first axis. 1-D arrays must have the same length.

**Returns**

**stacked**

[ndarray] The array formed by stacking the given arrays, will be at least 2-D.

**See also:**

**concatenate**

Join a sequence of arrays along an existing axis.

**stack**

Join a sequence of arrays along a new axis.

**block**

Assemble an nd-array from nested lists of blocks.

**hstack**

Stack arrays in sequence horizontally (column wise).

**dstack**

Stack arrays in sequence depth wise (along third axis).

**column_stack**

Stack 1-D arrays as columns into a 2-D array.

**vsplit**

Split an array into multiple sub-arrays vertically (row-wise).

### Examples

```
>>> a = np.array([1, 2, 3])
>>> b = np.array([2, 3, 4])
>>> np.vstack((a,b))
array([[1, 2, 3],
       [2, 3, 4]])
```

```
>>> a = np.array([[1], [2], [3]])
>>> b = np.array([[2], [3], [4]])
>>> np.vstack((a,b))
array([[1],
       [2],
       [3],
       [2],
       [3],
       [4]])
```

`numpy.`**`hstack`**`(`*tup*`)`

Stack arrays in sequence horizontally (column wise).

This is equivalent to concatenation along the second axis, except for 1-D arrays where it concatenates along the first axis. Rebuilds arrays divided by `hsplit`.

This function makes most sense for arrays with up to 3 dimensions. For instance, for pixel-data with a height (first axis), width (second axis), and r/g/b channels (third axis). The functions `concatenate`, `stack` and `block` provide more general stacking and concatenation operations.

**Parameters**

**tup**

[sequence of ndarrays] The arrays must have the same shape along all but the second axis, except 1-D arrays which can be any length.

**Returns**

**stacked**

[ndarray] The array formed by stacking the given arrays.

**See also:**

**`concatenate`**

Join a sequence of arrays along an existing axis.

**`stack`**

Join a sequence of arrays along a new axis.

**`block`**

Assemble an nd-array from nested lists of blocks.

**`vstack`**

Stack arrays in sequence vertically (row wise).

**`dstack`**

Stack arrays in sequence depth wise (along third axis).

**`column_stack`**

Stack 1-D arrays as columns into a 2-D array.

**`hsplit`**

Split an array into multiple sub-arrays horizontally (column-wise).

### Examples

```
>>> a = np.array((1,2,3))
>>> b = np.array((2,3,4))
>>> np.hstack((a,b))
array([1, 2, 3, 2, 3, 4])
>>> a = np.array([[1],[2],[3]])
>>> b = np.array([[2],[3],[4]])
>>> np.hstack((a,b))
array([[1, 2],
       [2, 3],
       [3, 4]])
```

`numpy.`**`dstack`**`(`*tup*`)`

Stack arrays in sequence depth wise (along third axis).

This is equivalent to concatenation along the third axis after 2-D arrays of shape *(M,N)* have been reshaped to *(M,N,1)* and 1-D arrays of shape *(N,)* have been reshaped to *(1,N,1)*. Rebuilds arrays divided by `dsplit`.

This function makes most sense for arrays with up to 3 dimensions. For instance, for pixel-data with a height (first axis), width (second axis), and r/g/b channels (third axis). The functions `concatenate`, `stack` and `block` provide more general stacking and concatenation operations.

**Parameters**

**tup**

[sequence of arrays] The arrays must have the same shape along all but the third axis. 1-D or 2-D arrays must have the same shape.

**Returns**

**stacked**

[ndarray] The array formed by stacking the given arrays, will be at least 3-D.

**See also:**

**`concatenate`**

Join a sequence of arrays along an existing axis.

**`stack`**

Join a sequence of arrays along a new axis.

**`block`**

Assemble an nd-array from nested lists of blocks.

**`vstack`**

Stack arrays in sequence vertically (row wise).

**`hstack`**

Stack arrays in sequence horizontally (column wise).

**`column_stack`**

Stack 1-D arrays as columns into a 2-D array.

**`dsplit`**

Split array along third axis.

## Examples

```
>>> a = np.array((1,2,3))
>>> b = np.array((2,3,4))
>>> np.dstack((a,b))
array([[[1, 2],
        [2, 3],
        [3, 4]]])
```

```
>>> a = np.array([[1],[2],[3]])
>>> b = np.array([[2],[3],[4]])
>>> np.dstack((a,b))
array([[[1, 2]],
       [[2, 3]],
       [[3, 4]]])
```

`numpy.`**`column_stack`**(*tup*)

Stack 1-D arrays as columns into a 2-D array.

Take a sequence of 1-D arrays and stack them as columns to make a single 2-D array. 2-D arrays are stacked as-is, just like with `hstack`. 1-D arrays are turned into 2-D columns first.

**Parameters**

**tup**

[sequence of 1-D or 2-D arrays.] Arrays to stack. All of them must have the same first dimension.

**Returns**

**stacked**

[2-D array] The array formed by stacking the given arrays.

**See also:**

`stack`, `hstack`, `vstack`, `concatenate`

### Examples

```
>>> a = np.array((1,2,3))
>>> b = np.array((2,3,4))
>>> np.column_stack((a,b))
array([[1, 2],
       [2, 3],
       [3, 4]])
```

## 4.2.7 Splitting arrays

| | |
|---|---|
| `split`(ary, indices_or_sections[, axis]) | Split an array into multiple sub-arrays as views into *ary*. |
| `array_split`(ary, indices_or_sections[, axis]) | Split an array into multiple sub-arrays. |
| `dsplit`(ary, indices_or_sections) | Split array into multiple sub-arrays along the 3rd axis (depth). |
| `hsplit`(ary, indices_or_sections) | Split an array into multiple sub-arrays horizontally (column-wise). |
| `vsplit`(ary, indices_or_sections) | Split an array into multiple sub-arrays vertically (row-wise). |

`numpy.`**`split`**(*ary*, *indices_or_sections*, *axis=0*)

Split an array into multiple sub-arrays as views into *ary*.

**Parameters**

**ary**

[ndarray] Array to be divided into sub-arrays.

**indices_or_sections**

[int or 1-D array] If *indices_or_sections* is an integer, N, the array will be divided into N equal arrays along *axis*. If such a split is not possible, an error is raised.

If *indices_or_sections* is a 1-D array of sorted integers, the entries indicate where along *axis* the array is split. For example, `[2, 3]` would, for `axis=0`, result in

- ary[:2]

- ary[2:3]
- ary[3:]

If an index exceeds the dimension of the array along *axis*, an empty sub-array is returned correspondingly.

**axis**

[int, optional] The axis along which to split, default is 0.

**Returns**

**sub-arrays**

[list of ndarrays] A list of sub-arrays as views into *ary*.

**Raises**

**ValueError**

If *indices_or_sections* is given as an integer, but a split does not result in equal division.

**See also:**

**`array_split`**

Split an array into multiple sub-arrays of equal or near-equal size. Does not raise an exception if an equal division cannot be made.

**`hsplit`**

Split array into multiple sub-arrays horizontally (column-wise).

**`vsplit`**

Split array into multiple sub-arrays vertically (row wise).

**`dsplit`**

Split array into multiple sub-arrays along the 3rd axis (depth).

**`concatenate`**

Join a sequence of arrays along an existing axis.

**`stack`**

Join a sequence of arrays along a new axis.

**`hstack`**

Stack arrays in sequence horizontally (column wise).

**`vstack`**

Stack arrays in sequence vertically (row wise).

**`dstack`**

Stack arrays in sequence depth wise (along third dimension).

### Examples

```
>>> x = np.arange(9.0)
>>> np.split(x, 3)
[array([0.,  1.,  2.]), array([3.,  4.,  5.]), array([6.,  7.,  8.])]
```

```
>>> x = np.arange(8.0)
>>> np.split(x, [3, 5, 6, 10])
[array([0.,  1.,  2.]),
 array([3.,  4.]),
 array([5.]),
 array([6.,  7.]),
 array([], dtype=float64)]
```

`numpy.`**`array_split`**(*ary*, *indices_or_sections*, *axis=0*)

Split an array into multiple sub-arrays.

Please refer to the `split` documentation. The only difference between these functions is that `array_split` allows *indices_or_sections* to be an integer that does *not* equally divide the axis. For an array of length l that should be split into n sections, it returns l % n sub-arrays of size l//n + 1 and the rest of size l/n.

**See also:**

***`split`***

Split array into multiple sub-arrays of equal size.

### Examples

```
>>> x = np.arange(8.0)
>>> np.array_split(x, 3)
    [array([0.,  1.,  2.]), array([3.,  4.,  5.]), array([6.,  7.])]
```

```
>>> x = np.arange(7.0)
>>> np.array_split(x, 3)
    [array([0.,  1.,  2.]), array([3.,  4.]), array([5.,  6.])]
```

`numpy.`**`dsplit`**(*ary*, *indices_or_sections*)

Split array into multiple sub-arrays along the 3rd axis (depth).

Please refer to the `split` documentation. `dsplit` is equivalent to `split` with `axis=2`, the array is always split along the third axis provided the array dimension is greater than or equal to 3.

**See also:**

***`split`***

Split an array into multiple sub-arrays of equal size.

### Examples

```
>>> x = np.arange(16.0).reshape(2, 2, 4)
>>> x
array([[[ 0.,   1.,   2.,   3.],
        [ 4.,   5.,   6.,   7.]],
       [[ 8.,   9.,  10.,  11.],
        [12.,  13.,  14.,  15.]]])
>>> np.dsplit(x, 2)
[array([[[ 0.,  1.],
        [ 4.,  5.]],
       [[ 8.,  9.],
        [12., 13.]]]), array([[[ 2.,  3.],
        [ 6.,  7.]],
       [[10., 11.],
        [14., 15.]]])]
>>> np.dsplit(x, np.array([3, 6]))
[array([[[ 0.,   1.,   2.],
        [ 4.,   5.,   6.]],
       [[ 8.,   9.,  10.],
        [12.,  13.,  14.]]]),
 array([[[ 3.],
        [ 7.]],
       [[11.],
        [15.]]]),
array([], shape=(2, 2, 0), dtype=float64)]
```

`numpy.`**`hsplit`**(*ary*, *indices_or_sections*)

Split an array into multiple sub-arrays horizontally (column-wise).

Please refer to the *`split`* documentation. *`hsplit`* is equivalent to *`split`* with `axis=1`, the array is always split along the second axis regardless of the array dimension.

**See also:**

***`split`***

Split an array into multiple sub-arrays of equal size.

### Examples

```
>>> x = np.arange(16.0).reshape(4, 4)
>>> x
array([[ 0.,   1.,   2.,   3.],
       [ 4.,   5.,   6.,   7.],
       [ 8.,   9.,  10.,  11.],
       [12.,  13.,  14.,  15.]])
>>> np.hsplit(x, 2)
[array([[  0.,   1.],
       [  4.,   5.],
       [  8.,   9.],
       [12.,  13.]]),
 array([[  2.,   3.],
       [  6.,   7.],
       [10.,  11.],
       [14.,  15.]])]
```

(continues on next page)

(continued from previous page)

```
>>> np.hsplit(x, np.array([3, 6]))
[array([[ 0.,   1.,   2.],
       [ 4.,   5.,   6.],
       [ 8.,   9.,  10.],
       [12.,  13.,  14.]]),
 array([[ 3.],
       [ 7.],
       [11.],
       [15.]]),
 array([], shape=(4, 0), dtype=float64)]
```

With a higher dimensional array the split is still along the second axis.

```
>>> x = np.arange(8.0).reshape(2, 2, 2)
>>> x
array([[[0.,  1.],
        [2.,  3.]],
       [[4.,  5.],
        [6.,  7.]]])
>>> np.hsplit(x, 2)
[array([[[0.,  1.]],
       [[4.,  5.]]]),
 array([[[2.,  3.]],
       [[6.,  7.]]])]
```

`numpy.`**`vsplit`**(*ary*, *indices_or_sections*)

Split an array into multiple sub-arrays vertically (row-wise).

Please refer to the `split` documentation. `vsplit` is equivalent to `split` with *axis=0* (default), the array is always split along the first axis regardless of the array dimension.

**See also:**

***`split`***

Split an array into multiple sub-arrays of equal size.

### Examples

```
>>> x = np.arange(16.0).reshape(4, 4)
>>> x
array([[ 0.,   1.,   2.,   3.],
       [ 4.,   5.,   6.,   7.],
       [ 8.,   9.,  10.,  11.],
       [12.,  13.,  14.,  15.]])
>>> np.vsplit(x, 2)
[array([[0., 1., 2., 3.],
       [4., 5., 6., 7.]]), array([[ 8.,  9., 10., 11.],
       [12., 13., 14., 15.]])]
>>> np.vsplit(x, np.array([3, 6]))
[array([[ 0.,  1.,  2.,  3.],
       [ 4.,  5.,  6.,  7.],
       [ 8.,  9., 10., 11.]]), array([[12., 13., 14., 15.]]), array([], shape=(0,␣
↪4), dtype=float64)]
```

With a higher dimensional array the split is still along the first axis.

```
>>> x = np.arange(8.0).reshape(2, 2, 2)
>>> x
array([[[0.,  1.],
        [2.,  3.]],
       [[4.,  5.],
        [6.,  7.]]])
>>> np.vsplit(x, 2)
[array([[[0., 1.],
        [2., 3.]]]), array([[[4., 5.],
        [6., 7.]]])]
```

### 4.2.8 Tiling arrays

| | |
|---|---|
| *tile*(A, reps) | Construct an array by repeating A the number of times given by reps. |
| *repeat*(a, repeats[, axis]) | Repeat elements of an array. |

`numpy.`**`tile`**`(`*A*, *reps*`)`

Construct an array by repeating A the number of times given by reps.

If *reps* has length `d`, the result will have dimension of `max(d, A.ndim)`.

If `A.ndim < d`, *A* is promoted to be d-dimensional by prepending new axes. So a shape (3,) array is promoted to (1, 3) for 2-D replication, or shape (1, 1, 3) for 3-D replication. If this is not the desired behavior, promote *A* to d-dimensions manually before calling this function.

If `A.ndim > d`, *reps* is promoted to *A*.ndim by pre-pending 1's to it. Thus for an *A* of shape (2, 3, 4, 5), a *reps* of (2, 2) is treated as (1, 1, 2, 2).

Note : Although tile may be used for broadcasting, it is strongly recommended to use numpy's broadcasting operations and functions.

**Parameters**

**A**

[array_like] The input array.

**reps**

[array_like] The number of repetitions of *A* along each axis.

**Returns**

**c**

[ndarray] The tiled output array.

**See also:**

***repeat***

Repeat elements of an array.

***broadcast_to***

Broadcast an array to a new shape

### Examples

```
>>> a = np.array([0, 1, 2])
>>> np.tile(a, 2)
array([0, 1, 2, 0, 1, 2])
>>> np.tile(a, (2, 2))
array([[0, 1, 2, 0, 1, 2],
       [0, 1, 2, 0, 1, 2]])
>>> np.tile(a, (2, 1, 2))
array([[[0, 1, 2, 0, 1, 2]],
       [[0, 1, 2, 0, 1, 2]]])
```

```
>>> b = np.array([[1, 2], [3, 4]])
>>> np.tile(b, 2)
array([[1, 2, 1, 2],
       [3, 4, 3, 4]])
>>> np.tile(b, (2, 1))
array([[1, 2],
       [3, 4],
       [1, 2],
       [3, 4]])
```

```
>>> c = np.array([1,2,3,4])
>>> np.tile(c,(4,1))
array([[1, 2, 3, 4],
       [1, 2, 3, 4],
       [1, 2, 3, 4],
       [1, 2, 3, 4]])
```

`numpy.`**`repeat`**(*a*, *repeats*, *axis=None*)

Repeat elements of an array.

**Parameters**

**a**

[array_like] Input array.

**repeats**

[int or array of ints] The number of repetitions for each element. *repeats* is broadcasted to fit the shape of the given axis.

**axis**

[int, optional] The axis along which to repeat values. By default, use the flattened input array, and return a flat output array.

**Returns**

**repeated_array**

[ndarray] Output array which has the same shape as *a*, except along the given axis.

**See also:**

***`tile`***

Tile an array.

### Examples

```
>>> np.repeat(3, 4)
array([3, 3, 3, 3])
>>> x = np.array([[1,2],[3,4]])
>>> np.repeat(x, 2)
array([1, 1, 2, 2, 3, 3, 4, 4])
>>> np.repeat(x, 3, axis=1)
array([[1, 1, 1, 2, 2, 2],
       [3, 3, 3, 4, 4, 4]])
>>> np.repeat(x, [1, 2], axis=0)
array([[1, 2],
       [3, 4],
       [3, 4]])
```

## 4.2.9 Adding and removing elements

| | |
|---|---|
| `delete`(arr, obj[, axis]) | Return a new array with sub-arrays along an axis deleted. |
| `insert`(arr, obj, values[, axis]) | Insert values along the given axis before the given indices. |
| `append`(arr, values[, axis]) | Append values to the end of an array. |
| `resize`(a, new_shape) | Return a new array with the specified shape. |
| `trim_zeros`(filt[, trim]) | Trim the leading and/or trailing zeros from a 1-D array or sequence. |
| `unique`(ar[, return_index, return_inverse, …]) | Find the unique elements of an array. |

`numpy.`**`delete`**(*arr*, *obj*, *axis=None*)

Return a new array with sub-arrays along an axis deleted. For a one dimensional array, this returns those entries not returned by *arr[obj]*.

**Parameters**

**arr**

[array_like] Input array.

**obj**

[slice, int or array of ints] Indicate indices of sub-arrays to remove along the specified axis.

Changed in version 1.19.0: Boolean indices are now treated as a mask of elements to remove, rather than being cast to the integers 0 and 1.

**axis**

[int, optional] The axis along which to delete the subarray defined by *obj*. If *axis* is None, *obj* is applied to the flattened array.

**Returns**

**out**

[ndarray] A copy of *arr* with the elements specified by *obj* removed. Note that `delete` does not occur in-place. If *axis* is None, *out* is a flattened array.

**See also:**

***insert***

Insert elements into an array.

***append***

Append elements at the end of an array.

## Notes

Often it is preferable to use a boolean mask. For example:

```
>>> arr = np.arange(12) + 1
>>> mask = np.ones(len(arr), dtype=bool)
>>> mask[[0,2,4]] = False
>>> result = arr[mask,...]
```

Is equivalent to *np.delete(arr, [0,2,4], axis=0)*, but allows further use of *mask*.

## Examples

```
>>> arr = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])
>>> arr
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12]])
>>> np.delete(arr, 1, 0)
array([[ 1,  2,  3,  4],
       [ 9, 10, 11, 12]])
```

```
>>> np.delete(arr, np.s_[::2], 1)
array([[ 2,  4],
       [ 6,  8],
       [10, 12]])
>>> np.delete(arr, [1,3,5], None)
array([ 1,  3,  5,  7,  8,  9, 10, 11, 12])
```

`numpy.`**`insert`**(*arr*, *obj*, *values*, *axis=None*)

Insert values along the given axis before the given indices.

**Parameters**

**arr**

[array_like] Input array.

**obj**

[int, slice or sequence of ints] Object that defines the index or indices before which *values* is inserted.

New in version 1.8.0.

Support for multiple insertions when *obj* is a single scalar or a sequence with one element (similar to calling insert multiple times).

**values**

[array_like] Values to insert into *arr*. If the type of *values* is different from that of *arr*, *values* is converted to the type of *arr*. *values* should be shaped so that `arr[...,obj,...] = values` is legal.

**axis**

[int, optional] Axis along which to insert *values*. If *axis* is None then *arr* is flattened first.

**Returns**

**out**

[ndarray] A copy of *arr* with *values* inserted. Note that `insert` does not occur in-place: a new array is returned. If *axis* is None, *out* is a flattened array.

**See also:**

**`append`**

Append elements at the end of an array.

**`concatenate`**

Join a sequence of arrays along an existing axis.

**`delete`**

Delete elements from an array.

## Notes

Note that for higher dimensional inserts *obj=0* behaves very different from *obj=[0]* just like *arr[:,0,:] = values* is different from *arr[:,[0],:] = values*.

## Examples

```
>>> a = np.array([[1, 1], [2, 2], [3, 3]])
>>> a
array([[1, 1],
       [2, 2],
       [3, 3]])
>>> np.insert(a, 1, 5)
array([1, 5, 1, ..., 2, 3, 3])
>>> np.insert(a, 1, 5, axis=1)
array([[1, 5, 1],
       [2, 5, 2],
       [3, 5, 3]])
```

Difference between sequence and scalars:

```
>>> np.insert(a, [1], [[1],[2],[3]], axis=1)
array([[1, 1, 1],
       [2, 2, 2],
       [3, 3, 3]])
>>> np.array_equal(np.insert(a, 1, [1, 2, 3], axis=1),
...                np.insert(a, [1], [[1],[2],[3]], axis=1))
True
```

```
>>> b = a.flatten()
>>> b
array([1, 1, 2, 2, 3, 3])
>>> np.insert(b, [2, 2], [5, 6])
array([1, 1, 5, ..., 2, 3, 3])
```

```
>>> np.insert(b, slice(2, 4), [5, 6])
array([1, 1, 5, ..., 2, 3, 3])
```

```
>>> np.insert(b, [2, 2], [7.13, False]) # type casting
array([1, 1, 7, ..., 2, 3, 3])
```

```
>>> x = np.arange(8).reshape(2, 4)
>>> idx = (1, 3)
>>> np.insert(x, idx, 999, axis=1)
array([[  0, 999,   1,   2, 999,   3],
       [  4, 999,   5,   6, 999,   7]])
```

`numpy.`**`append`**(*arr*, *values*, *axis=None*)

Append values to the end of an array.

**Parameters**

**arr**

[array_like] Values are appended to a copy of this array.

**values**

[array_like] These values are appended to a copy of *arr*. It must be of the correct shape (the same shape as *arr*, excluding *axis*). If *axis* is not specified, *values* can be any shape and will be flattened before use.

**axis**

[int, optional] The axis along which *values* are appended. If *axis* is not given, both *arr* and *values* are flattened before use.

**Returns**

**append**

[ndarray] A copy of *arr* with *values* appended to *axis*. Note that `append` does not occur in-place: a new array is allocated and filled. If *axis* is None, *out* is a flattened array.

**See also:**

**`insert`**

Insert elements into an array.

**`delete`**

Delete elements from an array.

### Examples

```
>>> np.append([1, 2, 3], [[4, 5, 6], [7, 8, 9]])
array([1, 2, 3, ..., 7, 8, 9])
```

When *axis* is specified, *values* must have the correct shape.

```
>>> np.append([[1, 2, 3], [4, 5, 6]], [[7, 8, 9]], axis=0)
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
>>> np.append([[1, 2, 3], [4, 5, 6]], [7, 8, 9], axis=0)
Traceback (most recent call last):
    ...
ValueError: all the input arrays must have same number of dimensions, but
the array at index 0 has 2 dimension(s) and the array at index 1 has 1
dimension(s)
```

`numpy.`**`resize`**(*a*, *new_shape*)

Return a new array with the specified shape.

If the new array is larger than the original array, then the new array is filled with repeated copies of *a*. Note that this behavior is different from a.resize(new_shape) which fills with zeros instead of repeated copies of *a*.

**Parameters**

**a**

[array_like] Array to be resized.

**new_shape**

[int or tuple of int] Shape of resized array.

**Returns**

**reshaped_array**

[ndarray] The new array is formed from the data in the old array, repeated if necessary to fill out the required number of elements. The data are repeated in the order that they are stored in memory.

**See also:**

***`ndarray.resize`***

resize an array in-place.

### Notes

Warning: This functionality does **not** consider axes separately, i.e. it does not apply interpolation/extrapolation. It fills the return array with the required number of elements, taken from *a* as they are laid out in memory, disregarding strides and axes. (This is in case the new shape is smaller. For larger, see above.) This functionality is therefore not suitable to resize images, or data where each axis represents a separate and distinct entity.

### Examples

```
>>> a=np.array([[0,1],[2,3]])
>>> np.resize(a,(2,3))
array([[0, 1, 2],
       [3, 0, 1]])
>>> np.resize(a,(1,4))
array([[0, 1, 2, 3]])
>>> np.resize(a,(2,4))
array([[0, 1, 2, 3],
       [0, 1, 2, 3]])
```

`numpy.`**`trim_zeros`**(*filt*, *trim='fb'*)

Trim the leading and/or trailing zeros from a 1-D array or sequence.

**Parameters**

**filt**

[1-D array or sequence] Input array.

**trim**

[str, optional] A string with 'f' representing trim from front and 'b' to trim from back. Default is 'fb', trim zeros from both front and back of the array.

**Returns**

**trimmed**

[1-D array or sequence] The result of trimming the input. The input data type is preserved.

### Examples

```
>>> a = np.array((0, 0, 0, 1, 2, 3, 0, 2, 1, 0))
>>> np.trim_zeros(a)
array([1, 2, 3, 0, 2, 1])
```

```
>>> np.trim_zeros(a, 'b')
array([0, 0, 0, ..., 0, 2, 1])
```

The input data type is preserved, list/tuple in means list/tuple out.

```
>>> np.trim_zeros([0, 1, 2, 0])
[1, 2]
```

`numpy.`**`unique`**(*ar*, *return_index=False*, *return_inverse=False*, *return_counts=False*, *axis=None*)

Find the unique elements of an array.

Returns the sorted unique elements of an array. There are three optional outputs in addition to the unique elements:

- the indices of the input array that give the unique values
- the indices of the unique array that reconstruct the input array
- the number of times each unique value comes up in the input array

**Parameters**

**ar**

[array_like] Input array. Unless *axis* is specified, this will be flattened if it is not already 1-D.

**return_index**

[bool, optional] If True, also return the indices of *ar* (along the specified axis, if provided, or in the flattened array) that result in the unique array.

**return_inverse**

[bool, optional] If True, also return the indices of the unique array (for the specified axis, if provided) that can be used to reconstruct *ar*.

**return_counts**

[bool, optional] If True, also return the number of times each unique item appears in *ar*.

New in version 1.9.0.

**axis**

[int or None, optional] The axis to operate on. If None, *ar* will be flattened. If an integer, the subarrays indexed by the given axis will be flattened and treated as the elements of a 1-D array with the dimension of the given axis, see the notes for more details. Object arrays or structured arrays that contain objects are not supported if the *axis* kwarg is used. The default is None.

New in version 1.13.0.

**Returns**

**unique**

[ndarray] The sorted unique values.

**unique_indices**

[ndarray, optional] The indices of the first occurrences of the unique values in the original array. Only provided if *return_index* is True.

**unique_inverse**

[ndarray, optional] The indices to reconstruct the original array from the unique array. Only provided if *return_inverse* is True.

**unique_counts**

[ndarray, optional] The number of times each of the unique values comes up in the original array. Only provided if *return_counts* is True.

New in version 1.9.0.

**See also:**

**`numpy.lib.arraysetops`**

Module with a number of other functions for performing set operations on arrays.

### Notes

When an axis is specified the subarrays indexed by the axis are sorted. This is done by making the specified axis the first dimension of the array (move the axis to the first dimension to keep the order of the other axes) and then flattening the subarrays in C order. The flattened subarrays are then viewed as a structured type with each element given a label, with the effect that we end up with a 1-D array of structured types that can be treated in the same way as any other 1-D array. The result is that the flattened subarrays are sorted in lexicographic order starting with the first element.

### Examples

```
>>> np.unique([1, 1, 2, 2, 3, 3])
array([1, 2, 3])
>>> a = np.array([[1, 1], [2, 3]])
>>> np.unique(a)
array([1, 2, 3])
```

Return the unique rows of a 2D array

```
>>> a = np.array([[1, 0, 0], [1, 0, 0], [2, 3, 4]])
>>> np.unique(a, axis=0)
array([[1, 0, 0], [2, 3, 4]])
```

Return the indices of the original array that give the unique values:

```
>>> a = np.array(['a', 'b', 'b', 'c', 'a'])
>>> u, indices = np.unique(a, return_index=True)
>>> u
array(['a', 'b', 'c'], dtype='<U1')
>>> indices
array([0, 1, 3])
>>> a[indices]
array(['a', 'b', 'c'], dtype='<U1')
```

Reconstruct the input array from the unique values:

```
>>> a = np.array([1, 2, 6, 4, 2, 3, 2])
>>> u, indices = np.unique(a, return_inverse=True)
>>> u
array([1, 2, 3, 4, 6])
>>> indices
array([0, 1, 4, 3, 1, 2, 1])
>>> u[indices]
array([1, 2, 6, 4, 2, 3, 2])
```

### 4.2.10 Rearranging elements

| | |
|---|---|
| *flip*(m[, axis]) | Reverse the order of elements in an array along the given axis. |
| *fliplr*(m) | Flip array in the left/right direction. |
| *flipud*(m) | Flip array in the up/down direction. |
| *reshape*(a, newshape[, order]) | Gives a new shape to an array without changing its data. |
| *roll*(a, shift[, axis]) | Roll array elements along a given axis. |
| *rot90*(m[, k, axes]) | Rotate an array by 90 degrees in the plane specified by axes. |

numpy.**flip**(*m*, *axis=None*)

Reverse the order of elements in an array along the given axis.

The shape of the array is preserved, but the elements are reordered.

New in version 1.12.0.

**Parameters**

**m**

[array_like] Input array.

**axis**

[None or int or tuple of ints, optional] Axis or axes along which to flip over. The default, axis=None, will flip over all of the axes of the input array. If axis is negative it counts from the last to the first axis.

If axis is a tuple of ints, flipping is performed on all of the axes specified in the tuple.

Changed in version 1.15.0: None and tuples of axes are supported

**Returns**

**out**

[array_like] A view of *m* with the entries of axis reversed. Since a view is returned, this operation is done in constant time.

**See also:**

***flipud***

Flip an array vertically (axis=0).

***fliplr***

Flip an array horizontally (axis=1).

### Notes

flip(m, 0) is equivalent to flipud(m).

flip(m, 1) is equivalent to fliplr(m).

flip(m, n) corresponds to `m[...,::-1,...]` with `::-1` at position n.

flip(m) corresponds to `m[::-1,::-1,...,::-1]` with `::-1` at all positions.

flip(m, (0, 1)) corresponds to `m[::-1,::-1,...]` with `::-1` at position 0 and position 1.

### Examples

```
>>> A = np.arange(8).reshape((2,2,2))
>>> A
array([[[0, 1],
        [2, 3]],
       [[4, 5],
        [6, 7]]])
>>> np.flip(A, 0)
array([[[4, 5],
        [6, 7]],
       [[0, 1],
        [2, 3]]])
>>> np.flip(A, 1)
array([[[2, 3],
        [0, 1]],
       [[6, 7],
        [4, 5]]])
>>> np.flip(A)
array([[[7, 6],
        [5, 4]],
       [[3, 2],
        [1, 0]]])
>>> np.flip(A, (0, 2))
array([[[5, 4],
        [7, 6]],
       [[1, 0],
        [3, 2]]])
>>> A = np.random.randn(3,4,5)
>>> np.all(np.flip(A,2) == A[:,:,::-1,...])
True
```

`numpy.`**`fliplr`**`(`*m*`)`

Flip array in the left/right direction.

Flip the entries in each row in the left/right direction. Columns are preserved, but appear in a different order than before.

**Parameters**

**m**

[array_like] Input array, must be at least 2-D.

**Returns**

**f**

[ndarray] A view of *m* with the columns reversed. Since a view is returned, this operation is $\mathcal{O}(1)$.

**See also:**

***flipud***

Flip array in the up/down direction.

***rot90***

Rotate array counterclockwise.

## Notes

Equivalent to m[:,::-1]. Requires the array to be at least 2-D.

## Examples

```
>>> A = np.diag([1.,2.,3.])
>>> A
array([[1.,  0.,  0.],
       [0.,  2.,  0.],
       [0.,  0.,  3.]])
>>> np.fliplr(A)
array([[0.,  0.,  1.],
       [0.,  2.,  0.],
       [3.,  0.,  0.]])
```

```
>>> A = np.random.randn(2,3,5)
>>> np.all(np.fliplr(A) == A[:,::-1,...])
True
```

`numpy.`**`flipud`**`(`*m*`)`

Flip array in the up/down direction.

Flip the entries in each column in the up/down direction. Rows are preserved, but appear in a different order than before.

**Parameters**

**m**

[array_like] Input array.

**Returns**

**out**

[array_like] A view of *m* with the rows reversed. Since a view is returned, this operation is $\mathcal{O}(1)$.

**See also:**

***fliplr***

Flip array in the left/right direction.

***rot90***

Rotate array counterclockwise.

## Notes

Equivalent to `m[::-1,...]`. Does not require the array to be two-dimensional.

## Examples

```
>>> A = np.diag([1.0, 2, 3])
>>> A
array([[1.,  0.,  0.],
       [0.,  2.,  0.],
       [0.,  0.,  3.]])
>>> np.flipud(A)
array([[0.,  0.,  3.],
       [0.,  2.,  0.],
       [1.,  0.,  0.]])
```

```
>>> A = np.random.randn(2,3,5)
>>> np.all(np.flipud(A) == A[::-1,...])
True
```

```
>>> np.flipud([1,2])
array([2, 1])
```

`numpy.`**`roll`**(*a*, *shift*, *axis=None*)

Roll array elements along a given axis.

Elements that roll beyond the last position are re-introduced at the first.

**Parameters**

**a**

[array_like] Input array.

**shift**

[int or tuple of ints] The number of places by which elements are shifted. If a tuple, then *axis* must be a tuple of the same size, and each of the given axes is shifted by the corresponding number. If an int while *axis* is a tuple of ints, then the same value is used for all given axes.

**axis**

[int or tuple of ints, optional] Axis or axes along which elements are shifted. By default, the array is flattened before shifting, after which the original shape is restored.

**Returns**

**res**

[ndarray] Output array, with the same shape as *a*.

**See also:**

*`rollaxis`*

Roll the specified axis backwards, until it lies in a given position.

### Notes

New in version 1.12.0.

Supports rolling over multiple dimensions simultaneously.

### Examples

```
>>> x = np.arange(10)
>>> np.roll(x, 2)
array([8, 9, 0, 1, 2, 3, 4, 5, 6, 7])
>>> np.roll(x, -2)
array([2, 3, 4, 5, 6, 7, 8, 9, 0, 1])
```

```
>>> x2 = np.reshape(x, (2,5))
>>> x2
array([[0, 1, 2, 3, 4],
       [5, 6, 7, 8, 9]])
>>> np.roll(x2, 1)
array([[9, 0, 1, 2, 3],
       [4, 5, 6, 7, 8]])
>>> np.roll(x2, -1)
array([[1, 2, 3, 4, 5],
       [6, 7, 8, 9, 0]])
>>> np.roll(x2, 1, axis=0)
array([[5, 6, 7, 8, 9],
       [0, 1, 2, 3, 4]])
>>> np.roll(x2, -1, axis=0)
array([[5, 6, 7, 8, 9],
       [0, 1, 2, 3, 4]])
>>> np.roll(x2, 1, axis=1)
array([[4, 0, 1, 2, 3],
       [9, 5, 6, 7, 8]])
>>> np.roll(x2, -1, axis=1)
array([[1, 2, 3, 4, 0],
       [6, 7, 8, 9, 5]])
```

`numpy.`**`rot90`**(*m*, *k=1*, *axes=(0, 1)*)

Rotate an array by 90 degrees in the plane specified by axes.

Rotation direction is from the first towards the second axis.

**Parameters**

**m**

[array_like] Array of two or more dimensions.

**k**

[integer] Number of times the array is rotated by 90 degrees.

**axes: (2,) array_like**

The array is rotated in the plane defined by the axes. Axes must be different.

New in version 1.12.0.

**Returns**

**y**

[ndarray] A rotated view of *m*.

**See also:**

**`flip`**

Reverse the order of elements in an array along the given axis.

**`fliplr`**

Flip an array horizontally.

**`flipud`**

Flip an array vertically.

## Notes

rot90(m, k=1, axes=(1,0)) is the reverse of rot90(m, k=1, axes=(0,1)) rot90(m, k=1, axes=(1,0)) is equivalent to rot90(m, k=-1, axes=(0,1))

## Examples

```
>>> m = np.array([[1,2],[3,4]], int)
>>> m
array([[1, 2],
       [3, 4]])
>>> np.rot90(m)
array([[2, 4],
       [1, 3]])
>>> np.rot90(m, 2)
array([[4, 3],
       [2, 1]])
>>> m = np.arange(8).reshape((2,2,2))
>>> np.rot90(m, 1, (1,2))
array([[[1, 3],
        [0, 2]],
       [[5, 7],
        [4, 6]]])
```

# 4.3 Binary operations

## 4.3.1 Elementwise bit operations

| | |
|---|---|
| `bitwise_and`(x1, x2, /[, out, where, …]) | Compute the bit-wise AND of two arrays element-wise. |
| `bitwise_or`(x1, x2, /[, out, where, casting, …]) | Compute the bit-wise OR of two arrays element-wise. |
| `bitwise_xor`(x1, x2, /[, out, where, …]) | Compute the bit-wise XOR of two arrays element-wise. |
| `invert`(x, /[, out, where, casting, order, …]) | Compute bit-wise inversion, or bit-wise NOT, element-wise. |
| `left_shift`(x1, x2, /[, out, where, casting, …]) | Shift the bits of an integer to the left. |
| `right_shift`(x1, x2, /[, out, where, …]) | Shift the bits of an integer to the right. |

numpy.**bitwise_and**(*x1*, *x2*, */*, *out=None*, ***, *where=True*, *casting='same_kind'*, *order='K'*, *dtype=None*, *subok=True*[, *signature*, *extobj* ]) = **<ufunc 'bitwise_and'>**

Compute the bit-wise AND of two arrays element-wise.

Computes the bit-wise AND of the underlying binary representation of the integers in the input arrays. This ufunc implements the C/Python operator `&`.

**Parameters**

**x1, x2**

[array_like] Only integer and boolean types are handled. If `x1.shape != x2.shape`, they must be broadcastable to a common shape (which becomes the shape of the output).

**out**

[ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

**where**

[array_like, optional] This condition is broadcast over the input. At locations where the condition is True, the *out* array will be set to the ufunc result. Elsewhere, the *out* array will retain its original value. Note that if an uninitialized *out* array is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.

****kwargs**

For other keyword-only arguments, see the *ufunc docs*.

**Returns**

**out**

[ndarray or scalar] Result. This is a scalar if both *x1* and *x2* are scalars.

**See also:**

`logical_and`, `bitwise_or`, `bitwise_xor`

**`binary_repr`**

Return the binary representation of the input number as a string.

### Examples

The number 13 is represented by `00001101`. Likewise, 17 is represented by `00010001`. The bit-wise AND of 13 and 17 is therefore `000000001`, or 1:

```
>>> np.bitwise_and(13, 17)
1
```

```
>>> np.bitwise_and(14, 13)
12
>>> np.binary_repr(12)
'1100'
>>> np.bitwise_and([14,3], 13)
array([12,  1])
```

```
>>> np.bitwise_and([11,7], [4,25])
array([0, 1])
>>> np.bitwise_and(np.array([2,5,255]), np.array([3,14,16]))
array([ 2,  4, 16])
>>> np.bitwise_and([True, True], [False, True])
array([False,  True])
```

`numpy.`**`bitwise_or`**(*x1*, *x2*, */*, *out=None*, ***, *where=True*, *casting='same_kind'*, *order='K'*, *dtype=None*, *subok=True*[, *signature*, *extobj* ]) **`= <ufunc 'bitwise_or'>`**

Compute the bit-wise OR of two arrays element-wise.

Computes the bit-wise OR of the underlying binary representation of the integers in the input arrays. This ufunc implements the C/Python operator `|`.

**Parameters**

**x1, x2**

[array_like] Only integer and boolean types are handled. If `x1.shape != x2.shape`, they must be broadcastable to a common shape (which becomes the shape of the output).

**out**

[ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

**where**

[array_like, optional] This condition is broadcast over the input. At locations where the condition is True, the *out* array will be set to the ufunc result. Elsewhere, the *out* array will retain its original value. Note that if an uninitialized *out* array is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.

****kwargs**

For other keyword-only arguments, see the *ufunc docs*.

**Returns**

**out**

[ndarray or scalar] Result. This is a scalar if both *x1* and *x2* are scalars.

**See also:**

*logical_or*, *bitwise_and*, *bitwise_xor*

***binary_repr***

Return the binary representation of the input number as a string.

## Examples

The number 13 has the binaray representation `00001101`. Likewise, 16 is represented by `00010000`. The bit-wise OR of 13 and 16 is then `000111011`, or 29:

```
>>> np.bitwise_or(13, 16)
29
>>> np.binary_repr(29)
'11101'
```

```
>>> np.bitwise_or(32, 2)
34
>>> np.bitwise_or([33, 4], 1)
array([33,  5])
>>> np.bitwise_or([33, 4], [1, 2])
array([33,  6])
```

```
>>> np.bitwise_or(np.array([2, 5, 255]), np.array([4, 4, 4]))
array([  6,   5, 255])
>>> np.array([2, 5, 255]) | np.array([4, 4, 4])
array([  6,   5, 255])
>>> np.bitwise_or(np.array([2, 5, 255, 2147483647], dtype=np.int32),
...               np.array([4, 4, 4, 2147483647], dtype=np.int32))
array([         6,          5,        255, 2147483647])
>>> np.bitwise_or([True, True], [False, True])
array([ True,  True])
```

`numpy.`**`bitwise_xor`**(*x1*, *x2*, */*, *out=None*, ***, *where=True*, *casting='same_kind'*, *order='K'*, *dtype=None*, *subok=True*[, *signature*, *extobj* ]) **`= <ufunc 'bitwise_xor'>`**

Compute the bit-wise XOR of two arrays element-wise.

Computes the bit-wise XOR of the underlying binary representation of the integers in the input arrays. This ufunc implements the C/Python operator `^`.

**Parameters**

**x1, x2**

[array_like] Only integer and boolean types are handled. If `x1.shape != x2.shape`, they must be broadcastable to a common shape (which becomes the shape of the output).

**out**

[ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

**where**

[array_like, optional] This condition is broadcast over the input. At locations where the condition is True, the *out* array will be set to the ufunc result. Elsewhere, the *out* array will retain

its original value. Note that if an uninitialized *out* array is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.

****kwargs**

For other keyword-only arguments, see the *ufunc docs*.

**Returns**

**out**

[ndarray or scalar] Result. This is a scalar if both *x1* and *x2* are scalars.

**See also:**

`logical_xor`, `bitwise_and`, `bitwise_or`

**`binary_repr`**

Return the binary representation of the input number as a string.

### Examples

The number 13 is represented by `00001101`. Likewise, 17 is represented by `00010001`. The bit-wise XOR of 13 and 17 is therefore `00011100`, or 28:

```
>>> np.bitwise_xor(13, 17)
28
>>> np.binary_repr(28)
'11100'
```

```
>>> np.bitwise_xor(31, 5)
26
>>> np.bitwise_xor([31,3], 5)
array([26,  6])
```

```
>>> np.bitwise_xor([31,3], [5,6])
array([26,  5])
>>> np.bitwise_xor([True, True], [False, True])
array([ True, False])
```

`numpy.`**`invert`**(*x*, */*, *out=None*, ***, *where=True*, *casting='same_kind'*, *order='K'*, *dtype=None*, *subok=True*[, *signature*, *extobj*]) **`= <ufunc 'invert'>`**

Compute bit-wise inversion, or bit-wise NOT, element-wise.

Computes the bit-wise NOT of the underlying binary representation of the integers in the input arrays. This ufunc implements the C/Python operator `~`.

For signed integer inputs, the two's complement is returned. In a two's-complement system negative numbers are represented by the two's complement of the absolute value. This is the most common method of representing signed integers on computers [1]. A N-bit two's-complement system can represent every integer in the range $-2^{N-1}$ to $+2^{N-1}-1$.

**Parameters**

**x**

[array_like] Only integer and boolean types are handled.

**out**

[ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

**where**

[array_like, optional] This condition is broadcast over the input. At locations where the condition is True, the *out* array will be set to the ufunc result. Elsewhere, the *out* array will retain its original value. Note that if an uninitialized *out* array is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.

****kwargs**

For other keyword-only arguments, see the *ufunc docs*.

**Returns**

**out**

[ndarray or scalar] Result. This is a scalar if *x* is a scalar.

**See also:**

`bitwise_and`, `bitwise_or`, `bitwise_xor`, `logical_not`

**`binary_repr`**

Return the binary representation of the input number as a string.

## Notes

`bitwise_not` is an alias for `invert`:

```
>>> np.bitwise_not is np.invert
True
```

## References

[1]

## Examples

We've seen that 13 is represented by `00001101`. The invert or bit-wise NOT of 13 is then:

```
>>> x = np.invert(np.array(13, dtype=np.uint8))
>>> x
242
>>> np.binary_repr(x, width=8)
'11110010'
```

The result depends on the bit-width:

```
>>> x = np.invert(np.array(13, dtype=np.uint16))
>>> x
65522
>>> np.binary_repr(x, width=16)
'1111111111110010'
```

When using signed integer types the result is the two's complement of the result for the unsigned type:

```
>>> np.invert(np.array([13], dtype=np.int8))
array([-14], dtype=int8)
>>> np.binary_repr(-14, width=8)
'11110010'
```

Booleans are accepted as well:

```
>>> np.invert(np.array([True, False]))
array([False,  True])
```

`numpy.`**`left_shift`**(*x1*, *x2*, /, *out=None*, *, *where=True*, *casting='same_kind'*, *order='K'*, *dtype=None*, *subok=True*[, *signature*, *extobj*]) **= <ufunc 'left_shift'>**

Shift the bits of an integer to the left.

Bits are shifted to the left by appending *x2* 0s at the right of *x1*. Since the internal representation of numbers is in binary format, this operation is equivalent to multiplying *x1* by `2**x2`.

**Parameters**

**x1**

[array_like of integer type] Input values.

**x2**

[array_like of integer type] Number of zeros to append to *x1*. Has to be non-negative. If `x1.shape != x2.shape`, they must be broadcastable to a common shape (which becomes the shape of the output).

**out**

[ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

**where**

[array_like, optional] This condition is broadcast over the input. At locations where the condition is True, the *out* array will be set to the ufunc result. Elsewhere, the *out* array will retain its original value. Note that if an uninitialized *out* array is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.

****kwargs**

For other keyword-only arguments, see the *ufunc docs*.

**Returns**

**out**

[array of integer type] Return *x1* with bits shifted *x2* times to the left. This is a scalar if both *x1* and *x2* are scalars.

**See also:**

*`right_shift`*

Shift the bits of an integer to the right.

*`binary_repr`*

Return the binary representation of the input number as a string.

### Examples

```
>>> np.binary_repr(5)
'101'
>>> np.left_shift(5, 2)
20
>>> np.binary_repr(20)
'10100'
```

```
>>> np.left_shift(5, [1,2,3])
array([10, 20, 40])
```

Note that the dtype of the second argument may change the dtype of the result and can lead to unexpected results in some cases (see *Casting Rules*):

```
>>> a = np.left_shift(np.uint8(255), 1) # Expect 254
>>> print(a, type(a)) # Unexpected result due to upcasting
510 <class 'numpy.int64'>
>>> b = np.left_shift(np.uint8(255), np.uint8(1))
>>> print(b, type(b))
254 <class 'numpy.uint8'>
```

`numpy.`**`right_shift`**(*x1*, *x2*, */*, *out=None*, ***, *where=True*, *casting='same_kind'*, *order='K'*, *dtype=None*, *subok=True*[, *signature*, *extobj*]) **`= <ufunc 'right_shift'>`**

Shift the bits of an integer to the right.

Bits are shifted to the right *x2*. Because the internal representation of numbers is in binary format, this operation is equivalent to dividing *x1* by `2**x2`.

**Parameters**

**x1**

[array_like, int] Input values.

**x2**

[array_like, int] Number of bits to remove at the right of *x1*. If `x1.shape != x2.shape`, they must be broadcastable to a common shape (which becomes the shape of the output).

**out**

[ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

**where**

[array_like, optional] This condition is broadcast over the input. At locations where the condition is True, the *out* array will be set to the ufunc result. Elsewhere, the *out* array will retain its original value. Note that if an uninitialized *out* array is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.

****kwargs**

For other keyword-only arguments, see the *ufunc docs*.

**Returns**

**out**

[ndarray, int] Return *x1* with bits shifted *x2* times to the right. This is a scalar if both *x1* and *x2* are scalars.

**See also:**

***left_shift***

Shift the bits of an integer to the left.

***binary_repr***

Return the binary representation of the input number as a string.

### Examples

```
>>> np.binary_repr(10)
'1010'
>>> np.right_shift(10, 1)
5
>>> np.binary_repr(5)
'101'
```

```
>>> np.right_shift(10, [1,2,3])
array([5, 2, 1])
```

## 4.3.2 Bit packing

| | |
|---|---|
| `packbits`(a[, axis, bitorder]) | Packs the elements of a binary-valued array into bits in a uint8 array. |
| `unpackbits`(a[, axis, count, bitorder]) | Unpacks elements of a uint8 array into a binary-valued output array. |

`numpy.`**`packbits`**(*a*, *axis=None*, *bitorder='big'*)

Packs the elements of a binary-valued array into bits in a uint8 array.

The result is padded to full bytes by inserting zero bits at the end.

**Parameters**

**a**

[array_like] An array of integers or booleans whose elements should be packed to bits.

**axis**

[int, optional] The dimension over which bit-packing is done. `None` implies packing the flattened array.

**bitorder**

[{'big', 'little'}, optional] The order of the input bits. 'big' will mimic bin(val), `[0, 0, 0, 0, 0, 0, 1, 1] => 3 = 0b00000011`, 'little' will reverse the order so `[1, 1, 0, 0, 0, 0, 0, 0] => 3`. Defaults to 'big'.

New in version 1.17.0.

**Returns**

**packed**

[ndarray] Array of type uint8 whose elements represent bits corresponding to the logical (0 or nonzero) value of the input elements. The shape of *packed* has the same number of dimensions as the input (unless *axis* is None, in which case the output is 1-D).

**See also:**

***unpackbits***

Unpacks elements of a uint8 array into a binary-valued output array.

## Examples

```
>>> a = np.array([[[1,0,1],
...                [0,1,0]],
...               [[1,1,0],
...                [0,0,1]]])
>>> b = np.packbits(a, axis=-1)
>>> b
array([[[160],
        [ 64]],
       [[192],
        [ 32]]], dtype=uint8)
```

Note that in binary 160 = 1010 0000, 64 = 0100 0000, 192 = 1100 0000, and 32 = 0010 0000.

`numpy.`**`unpackbits`**(*a*, *axis=None*, *count=None*, *bitorder='big'*)

Unpacks elements of a uint8 array into a binary-valued output array.

Each element of *a* represents a bit-field that should be unpacked into a binary-valued output array. The shape of the output array is either 1-D (if *axis* is `None`) or the same shape as the input array with unpacking done along the axis specified.

**Parameters**

**a**

[ndarray, uint8 type] Input array.

**axis**

[int, optional] The dimension over which bit-unpacking is done. `None` implies unpacking the flattened array.

**count**

[int or None, optional] The number of elements to unpack along *axis*, provided as a way of undoing the effect of packing a size that is not a multiple of eight. A non-negative number means to only unpack *count* bits. A negative number means to trim off that many bits from the end. `None` means to unpack the entire array (the default). Counts larger than the available number of bits will add zero padding to the output. Negative counts must not exceed the available number of bits.

New in version 1.17.0.

**bitorder**

[{'big', 'little'}, optional] The order of the returned bits. 'big' will mimic bin(val), `3 = 0b00000011 => [0, 0, 0, 0, 0, 0, 1, 1]`, 'little' will reverse the order to `[1, 1, 0, 0, 0, 0, 0, 0]`. Defaults to 'big'.

New in version 1.17.0.

**Returns**

**unpacked**

[ndarray, uint8 type] The elements are binary-valued (0 or 1).

**See also:**

**`packbits`**

Packs the elements of a binary-valued array into bits in a uint8 array.

## Examples

```
>>> a = np.array([[2], [7], [23]], dtype=np.uint8)
>>> a
array([[ 2],
       [ 7],
       [23]], dtype=uint8)
>>> b = np.unpackbits(a, axis=1)
>>> b
array([[0, 0, 0, 0, 0, 0, 1, 0],
       [0, 0, 0, 0, 0, 1, 1, 1],
       [0, 0, 0, 1, 0, 1, 1, 1]], dtype=uint8)
>>> c = np.unpackbits(a, axis=1, count=-3)
>>> c
array([[0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0],
       [0, 0, 0, 1, 0]], dtype=uint8)
```

```
>>> p = np.packbits(b, axis=0)
>>> np.unpackbits(p, axis=0)
array([[0, 0, 0, 0, 0, 0, 1, 0],
       [0, 0, 0, 0, 0, 1, 1, 1],
       [0, 0, 0, 1, 0, 1, 1, 1],
       [0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0],
```

(continues on next page)

(continued from previous page)

```
       [0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0]], dtype=uint8)
>>> np.array_equal(b, np.unpackbits(p, axis=0, count=b.shape[0]))
True
```

### 4.3.3 Output formatting

| | |
|---|---|
| *binary_repr*(num[, width]) | Return the binary representation of the input number as a string. |

`numpy.`**`binary_repr`**(*num*, *width=None*)

Return the binary representation of the input number as a string.

For negative numbers, if width is not given, a minus sign is added to the front. If width is given, the two's complement of the number is returned, with respect to that width.

In a two's-complement system negative numbers are represented by the two's complement of the absolute value. This is the most common method of representing signed integers on computers [1]. A N-bit two's-complement system can represent every integer in the range $-2^{N-1}$ to $+2^{N-1}-1$.

**Parameters**

**num**

[int] Only an integer decimal number can be used.

**width**

[int, optional] The length of the returned string if *num* is positive, or the length of the two's complement if *num* is negative, provided that *width* is at least a sufficient number of bits for *num* to be represented in the designated form.

If the *width* value is insufficient, it will be ignored, and *num* will be returned in binary (*num* > 0) or two's complement (*num* < 0) form with its width equal to the minimum number of bits needed to represent the number in the designated form. This behavior is deprecated and will later raise an error.

Deprecated since version 1.12.0.

**Returns**

**bin**

[str] Binary representation of *num* or two's complement of *num*.

**See also:**

**`base_repr`**

Return a string representation of a number in the given base system.

**`bin`**

Python's built-in binary representation generator of an integer.

### Notes

*binary_repr* is equivalent to using *base_repr* with base 2, but about 25x faster.

### References

[1]

### Examples

```
>>> np.binary_repr(3)
'11'
>>> np.binary_repr(-3)
'-11'
>>> np.binary_repr(3, width=4)
'0011'
```

The two's complement is returned when the input number is negative and width is specified:

```
>>> np.binary_repr(-3, width=3)
'101'
>>> np.binary_repr(-3, width=5)
'11101'
```

# 4.4 String operations

The *numpy.char* module provides a set of vectorized string operations for arrays of type `numpy.string_` or `numpy.unicode_`. All of them are based on the string methods in the Python standard library.

## 4.4.1 String operations

| | |
|---|---|
| *add*(x1, x2) | Return element-wise string concatenation for two arrays of str or unicode. |
| *multiply*(a, i) | Return (a * i), that is string multiple concatenation, element-wise. |
| *mod*(a, values) | Return (a % i), that is pre-Python 2.6 string formatting (interpolation), element-wise for a pair of array_likes of str or unicode. |
| *capitalize*(a) | Return a copy of *a* with only the first character of each element capitalized. |
| *center*(a, width[, fillchar]) | Return a copy of *a* with its elements centered in a string of length *width*. |
| *decode*(a[, encoding, errors]) | Calls *str.decode* element-wise. |
| *encode*(a[, encoding, errors]) | Calls *str.encode* element-wise. |
| *expandtabs*(a[, tabsize]) | Return a copy of each string element where all tab characters are replaced by one or more spaces. |
| *join*(sep, seq) | Return a string which is the concatenation of the strings in the sequence *seq*. |

Continued on next page

Table 21 – continued from previous page

| | |
|---|---|
| `ljust`(a, width[, fillchar]) | Return an array with the elements of *a* left-justified in a string of length *width*. |
| `lower`(a) | Return an array with the elements converted to lowercase. |
| `lstrip`(a[, chars]) | For each element in *a*, return a copy with the leading characters removed. |
| `partition`(a, sep) | Partition each element in *a* around *sep*. |
| `replace`(a, old, new[, count]) | For each element in *a*, return a copy of the string with all occurrences of substring *old* replaced by *new*. |
| `rjust`(a, width[, fillchar]) | Return an array with the elements of *a* right-justified in a string of length *width*. |
| `rpartition`(a, sep) | Partition (split) each element around the right-most separator. |
| `rsplit`(a[, sep, maxsplit]) | For each element in *a*, return a list of the words in the string, using *sep* as the delimiter string. |
| `rstrip`(a[, chars]) | For each element in *a*, return a copy with the trailing characters removed. |
| `split`(a[, sep, maxsplit]) | For each element in *a*, return a list of the words in the string, using *sep* as the delimiter string. |
| `splitlines`(a[, keepends]) | For each element in *a*, return a list of the lines in the element, breaking at line boundaries. |
| `strip`(a[, chars]) | For each element in *a*, return a copy with the leading and trailing characters removed. |
| `swapcase`(a) | Return element-wise a copy of the string with uppercase characters converted to lowercase and vice versa. |
| `title`(a) | Return element-wise title cased version of string or unicode. |
| `translate`(a, table[, deletechars]) | For each element in *a*, return a copy of the string where all characters occurring in the optional argument *deletechars* are removed, and the remaining characters have been mapped through the given translation table. |
| `upper`(a) | Return an array with the elements converted to uppercase. |
| `zfill`(a, width) | Return the numeric string left-filled with zeros |

`numpy.char.`**`add`**`(`*x1*, *x2*`)`

Return element-wise string concatenation for two arrays of str or unicode.

Arrays *x1* and *x2* must have the same shape.

**Parameters**

**x1**

[array_like of str or unicode] Input array.

**x2**

[array_like of str or unicode] Input array.

**Returns**

**add**

[ndarray] Output array of `string_` or `unicode_`, depending on input types of the same shape as *x1* and *x2*.

`numpy.char.`**`multiply`**`(`*a*, *i*`)`

Return (a * i), that is string multiple concatenation, element-wise.

Values in *i* of less than 0 are treated as 0 (which yields an empty string).

**Parameters**

**a**

[array_like of str or unicode]

**i**

[array_like of ints]

**Returns**

**out**

[ndarray] Output array of str or unicode, depending on input types

`numpy.char.`**`mod`**`(`*a*, *values*`)`

Return (a % i), that is pre-Python 2.6 string formatting (interpolation), element-wise for a pair of array_likes of str or unicode.

**Parameters**

**a**

[array_like of str or unicode]

**values**

[array_like of values] These values will be element-wise interpolated into the string.

**Returns**

**out**

[ndarray] Output array of str or unicode, depending on input types

**See also:**

`str.__mod__`

`numpy.char.`**`capitalize`**`(`*a*`)`

Return a copy of *a* with only the first character of each element capitalized.

Calls *str.capitalize* element-wise.

For 8-bit strings, this method is locale-dependent.

**Parameters**

**a**

[array_like of str or unicode] Input array of strings to capitalize.

**Returns**

**out**

[ndarray] Output array of str or unicode, depending on input types

**See also:**

`str.capitalize`

### Examples

```
>>> c = np.array(['a1b2','1b2a','b2a1','2a1b'],'S4'); c
array(['a1b2', '1b2a', 'b2a1', '2a1b'],
    dtype='|S4')
>>> np.char.capitalize(c)
array(['A1b2', '1b2a', 'B2a1', '2a1b'],
    dtype='|S4')
```

`numpy.char.`**`center`**(*a*, *width*, *fillchar=' '*)

Return a copy of *a* with its elements centered in a string of length *width*.

Calls *str.center* element-wise.

**Parameters**

**a**

[array_like of str or unicode]

**width**

[int] The length of the resulting strings

**fillchar**

[str or unicode, optional] The padding character to use (default is space).

**Returns**

**out**

[ndarray] Output array of str or unicode, depending on input types

**See also:**

`str.center`

`numpy.char.`**`decode`**(*a*, *encoding=None*, *errors=None*)

Calls *str.decode* element-wise.

The set of available codecs comes from the Python standard library, and may be extended at runtime. For more information, see the `codecs` module.

**Parameters**

**a**

[array_like of str or unicode]

**encoding**

[str, optional] The name of an encoding

**errors**

[str, optional] Specifies how to handle encoding errors

**Returns**

**out**

[ndarray]

**See also:**

`str.decode`

## Notes

The type of the result will depend on the encoding specified.

## Examples

```
>>> c = np.array(['aAaAaA', '  aA  ', 'abBABba'])
>>> c
array(['aAaAaA', '  aA  ', 'abBABba'], dtype='<U7')
>>> np.char.encode(c, encoding='cp037')
array(['\x81\xc1\x81\xc1\x81\xc1', '@@\x81\xc1@@',
    '\x81\x82\xc2\xc1\xc2\x82\x81'],
    dtype='|S7')
```

`numpy.char.`**`encode`**(*a*, *encoding=None*, *errors=None*)

Calls *str.encode* element-wise.

The set of available codecs comes from the Python standard library, and may be extended at runtime. For more information, see the codecs module.

**Parameters**

**a**

[array_like of str or unicode]

**encoding**

[str, optional] The name of an encoding

**errors**

[str, optional] Specifies how to handle encoding errors

**Returns**

**out**

[ndarray]

**See also:**

`str.encode`

### Notes

The type of the result will depend on the encoding specified.

`numpy.char.`**`expandtabs`**`(`*a*, *tabsize=8*`)`

Return a copy of each string element where all tab characters are replaced by one or more spaces.

Calls *str.expandtabs* element-wise.

Return a copy of each string element where all tab characters are replaced by one or more spaces, depending on the current column and the given *tabsize*. The column number is reset to zero after each newline occurring in the string. This doesn't understand other non-printing characters or escape sequences.

**Parameters**

**a**

[array_like of str or unicode] Input array

**tabsize**

[int, optional] Replace tabs with *tabsize* number of spaces. If not given defaults to 8 spaces.

**Returns**

**out**

[ndarray] Output array of str or unicode, depending on input type

**See also:**

`str.expandtabs`

`numpy.char.`**`join`**`(`*sep*, *seq*`)`

Return a string which is the concatenation of the strings in the sequence *seq*.

Calls *str.join* element-wise.

**Parameters**

**sep**

[array_like of str or unicode]

**seq**

[array_like of str or unicode]

**Returns**

**out**

[ndarray] Output array of str or unicode, depending on input types

**See also:**

`str.join`

`numpy.char.`**`ljust`**`(`*a*, *width*, *fillchar=' '*`)`

Return an array with the elements of *a* left-justified in a string of length *width*.

Calls *str.ljust* element-wise.

**Parameters**

**a**

[array_like of str or unicode]

**width**

[int] The length of the resulting strings

**fillchar**

[str or unicode, optional] The character to use for padding

**Returns**

**out**

[ndarray] Output array of str or unicode, depending on input type

**See also:**

`str.ljust`

`numpy.char.`**`lower`**`(`*a*`)`

Return an array with the elements converted to lowercase.

Call *str.lower* element-wise.

For 8-bit strings, this method is locale-dependent.

**Parameters**

**a**

[array_like, {str, unicode}] Input array.

**Returns**

**out**

[ndarray, {str, unicode}] Output array of str or unicode, depending on input type

**See also:**

`str.lower`

## Examples

```
>>> c = np.array(['A1B C', '1BCA', 'BCA1']); c
array(['A1B C', '1BCA', 'BCA1'], dtype='<U5')
>>> np.char.lower(c)
array(['a1b c', '1bca', 'bca1'], dtype='<U5')
```

`numpy.char.`**`lstrip`**`(`*a*, *chars=None*`)`

For each element in *a*, return a copy with the leading characters removed.

Calls *str.lstrip* element-wise.

**Parameters**

**a**

[array-like, {str, unicode}] Input array.

**chars**

[{str, unicode}, optional] The *chars* argument is a string specifying the set of characters to be removed. If omitted or None, the *chars* argument defaults to removing whitespace. The *chars* argument is not a prefix; rather, all combinations of its values are stripped.

**Returns**

**out**

[ndarray, {str, unicode}] Output array of str or unicode, depending on input type

**See also:**

`str.lstrip`

## Examples

```
>>> c = np.array(['aAaAaA', '  aA  ', 'abBABba'])
>>> c
array(['aAaAaA', '  aA  ', 'abBABba'], dtype='<U7')
```

The 'a' variable is unstripped from c[1] because whitespace leading.

```
>>> np.char.lstrip(c, 'a')
array(['AaAaA', '  aA  ', 'bBABba'], dtype='<U7')
```

```
>>> np.char.lstrip(c, 'A') # leaves c unchanged
array(['aAaAaA', '  aA  ', 'abBABba'], dtype='<U7')
>>> (np.char.lstrip(c, ' ') == np.char.lstrip(c, '')).all()
... # XXX: is this a regression? This used to return True
... # np.char.lstrip(c,'') does not modify c at all.
False
>>> (np.char.lstrip(c, ' ') == np.char.lstrip(c, None)).all()
True
```

`numpy.char.`**`partition`**`(`*a*, *sep*`)`

Partition each element in *a* around *sep*.

Calls *str.partition* element-wise.

For each element in *a*, split the element as the first occurrence of *sep*, and return 3 strings containing the part before the separator, the separator itself, and the part after the separator. If the separator is not found, return 3 strings containing the string itself, followed by two empty strings.

**Parameters**

**a**

[array_like, {str, unicode}] Input array

**sep**

[{str, unicode}] Separator to split each string element in *a*.

**Returns**

**out**

[ndarray, {str, unicode}] Output array of str or unicode, depending on input type. The output array will have an extra dimension with 3 elements per input element.

**See also:**

`str.partition`

`numpy.char.`**`replace`**`(`*a*, *old*, *new*, *count=None*`)`

For each element in *a*, return a copy of the string with all occurrences of substring *old* replaced by *new*.

Calls *str.replace* element-wise.

**Parameters**

**a**

[array-like of str or unicode]

**old, new**

[str or unicode]

**count**

[int, optional] If the optional argument `count` is given, only the first `count` occurrences are replaced.

**Returns**

**out**

[ndarray] Output array of str or unicode, depending on input type

**See also:**

`str.replace`

`numpy.char.`**`rjust`**`(`*a*, *width*, *fillchar=' '*`)`

Return an array with the elements of *a* right-justified in a string of length *width*.

Calls *str.rjust* element-wise.

**Parameters**

**a**

[array_like of str or unicode]

**width**

[int] The length of the resulting strings

**fillchar**

[str or unicode, optional] The character to use for padding

**Returns**

**out**

[ndarray] Output array of str or unicode, depending on input type

**See also:**

`str.rjust`

`numpy.char.`**`rpartition`**`(`*a*, *sep*`)`

Partition (split) each element around the right-most separator.

Calls *str.rpartition* element-wise.

For each element in *a*, split the element as the last occurrence of *sep*, and return 3 strings containing the part before the separator, the separator itself, and the part after the separator. If the separator is not found, return 3 strings containing the string itself, followed by two empty strings.

**Parameters**

**a**

[array_like of str or unicode] Input array

**sep**

[str or unicode] Right-most separator to split each element in array.

**Returns**

**out**

[ndarray] Output array of string or unicode, depending on input type. The output array will have an extra dimension with 3 elements per input element.

**See also:**

`str.rpartition`

`numpy.char.`**`rsplit`**`(`*a*, *sep=None*, *maxsplit=None*`)`

For each element in *a*, return a list of the words in the string, using *sep* as the delimiter string.

Calls *str.rsplit* element-wise.

Except for splitting from the right, *`rsplit`* behaves like *`split`*.

**Parameters**

**a**

[array_like of str or unicode]

**sep**

[str or unicode, optional] If *sep* is not specified or None, any whitespace string is a separator.

**maxsplit**

[int, optional] If *maxsplit* is given, at most *maxsplit* splits are done, the rightmost ones.

**Returns**

**out**

[ndarray] Array of list objects

**See also:**

`str.rsplit`, *`split`*

`numpy.char.`**`rstrip`**(*a*, *chars=None*)

For each element in *a*, return a copy with the trailing characters removed.

Calls *str.rstrip* element-wise.

**Parameters**

**a**

[array-like of str or unicode]

**chars**

[str or unicode, optional] The *chars* argument is a string specifying the set of characters to be removed. If omitted or None, the *chars* argument defaults to removing whitespace. The *chars* argument is not a suffix; rather, all combinations of its values are stripped.

**Returns**

**out**

[ndarray] Output array of str or unicode, depending on input type

**See also:**

`str.rstrip`

### Examples

```
>>> c = np.array(['aAaAaA', 'abBABba'], dtype='S7'); c
array(['aAaAaA', 'abBABba'],
    dtype='|S7')
>>> np.char.rstrip(c, b'a')
array(['aAaAaA', 'abBABb'],
    dtype='|S7')
>>> np.char.rstrip(c, b'A')
array(['aAaAa', 'abBABba'],
    dtype='|S7')
```

`numpy.char.`**`split`**(*a*, *sep=None*, *maxsplit=None*)

For each element in *a*, return a list of the words in the string, using *sep* as the delimiter string.

Calls *str.split* element-wise.

**Parameters**

**a**

[array_like of str or unicode]

**sep**

[str or unicode, optional] If *sep* is not specified or None, any whitespace string is a separator.

**maxsplit**

[int, optional] If *maxsplit* is given, at most *maxsplit* splits are done.

**Returns**

**out**

[ndarray] Array of list objects

**See also:**

`str.split`, *`rsplit`*

`numpy.char.`**`splitlines`**`(`*a*, *keepends=None*`)`

For each element in *a*, return a list of the lines in the element, breaking at line boundaries.

Calls *str.splitlines* element-wise.

**Parameters**

**a**

[array_like of str or unicode]

**keepends**

[bool, optional] Line breaks are not included in the resulting list unless keepends is given and true.

**Returns**

**out**

[ndarray] Array of list objects

**See also:**

`str.splitlines`

`numpy.char.`**`strip`**`(`*a*, *chars=None*`)`

For each element in *a*, return a copy with the leading and trailing characters removed.

Calls *str.strip* element-wise.

**Parameters**

**a**

[array-like of str or unicode]

**chars**

[str or unicode, optional] The *chars* argument is a string specifying the set of characters to be removed. If omitted or None, the *chars* argument defaults to removing whitespace. The *chars* argument is not a prefix or suffix; rather, all combinations of its values are stripped.

**Returns**

**out**

[ndarray] Output array of str or unicode, depending on input type

**See also:**

`str.strip`

## Examples

```
>>> c = np.array(['aAaAaA', '  aA  ', 'abBABba'])
>>> c
array(['aAaAaA', '  aA  ', 'abBABba'], dtype='<U7')
>>> np.char.strip(c)
array(['aAaAaA', 'aA', 'abBABba'], dtype='<U7')
>>> np.char.strip(c, 'a') # 'a' unstripped from c[1] because whitespace leads
array(['AaAaA', '  aA  ', 'bBABb'], dtype='<U7')
>>> np.char.strip(c, 'A') # 'A' unstripped from c[1] because (unprinted) ws trails
array(['aAaAa', '  aA  ', 'abBABba'], dtype='<U7')
```

`numpy.char.`**`swapcase`**`(`*a*`)`

Return element-wise a copy of the string with uppercase characters converted to lowercase and vice versa.

Calls *str.swapcase* element-wise.

For 8-bit strings, this method is locale-dependent.

**Parameters**

**a**

[array_like, {str, unicode}] Input array.

**Returns**

**out**

[ndarray, {str, unicode}] Output array of str or unicode, depending on input type

**See also:**

`str.swapcase`

## Examples

```
>>> c=np.array(['a1B c','1b Ca','b Ca1','cA1b'],'S5'); c
array(['a1B c', '1b Ca', 'b Ca1', 'cA1b'],
    dtype='|S5')
>>> np.char.swapcase(c)
array(['A1b C', '1B cA', 'B cA1', 'Ca1B'],
    dtype='|S5')
```

`numpy.char.`**`title`**`(`*a*`)`

Return element-wise title cased version of string or unicode.

Title case words start with uppercase characters, all remaining cased characters are lowercase.

Calls *str.title* element-wise.

For 8-bit strings, this method is locale-dependent.

**Parameters**

**a**

[array_like, {str, unicode}] Input array.

**Returns**

**out**

[ndarray] Output array of str or unicode, depending on input type

**See also:**

`str.title`

## Examples

```
>>> c=np.array(['a1b c','1b ca','b ca1','ca1b'],'S5'); c
array(['a1b c', '1b ca', 'b ca1', 'ca1b'],
    dtype='|S5')
>>> np.char.title(c)
array(['A1B C', '1B Ca', 'B Ca1', 'Ca1B'],
    dtype='|S5')
```

`numpy.char.`**`translate`**(*a*, *table*, *deletechars=None*)

For each element in *a*, return a copy of the string where all characters occurring in the optional argument *deletechars* are removed, and the remaining characters have been mapped through the given translation table.

Calls *str.translate* element-wise.

**Parameters**

**a**

[array-like of str or unicode]

**table**

[str of length 256]

**deletechars**

[str]

**Returns**

**out**

[ndarray] Output array of str or unicode, depending on input type

**See also:**

`str.translate`

`numpy.char.`**`upper`**(*a*)

Return an array with the elements converted to uppercase.

Calls *str.upper* element-wise.

For 8-bit strings, this method is locale-dependent.

**Parameters**

**a**

[array_like, {str, unicode}] Input array.

**Returns**

**out**

[ndarray, {str, unicode}] Output array of str or unicode, depending on input type

**See also:**

`str.upper`

## Examples

```
>>> c = np.array(['a1b c', '1bca', 'bca1']); c
array(['a1b c', '1bca', 'bca1'], dtype='<U5')
>>> np.char.upper(c)
array(['A1B C', '1BCA', 'BCA1'], dtype='<U5')
```

`numpy.char.`**`zfill`**`(`*a*, *width*`)`

Return the numeric string left-filled with zeros

Calls *str.zfill* element-wise.

**Parameters**

**a**

[array_like, {str, unicode}] Input array.

**width**

[int] Width of string to left-fill elements in *a*.

**Returns**

**out**

[ndarray, {str, unicode}] Output array of str or unicode, depending on input type

**See also:**

`str.zfill`

## 4.4.2 Comparison

Unlike the standard numpy comparison operators, the ones in the *char* module strip trailing whitespace characters before performing the comparison.

| | |
|---|---|
| `equal`(x1, x2) | Return (x1 == x2) element-wise. |
| `not_equal`(x1, x2) | Return (x1 != x2) element-wise. |
| `greater_equal`(x1, x2) | Return (x1 >= x2) element-wise. |
| `less_equal`(x1, x2) | Return (x1 <= x2) element-wise. |
| `greater`(x1, x2) | Return (x1 > x2) element-wise. |
| `less`(x1, x2) | Return (x1 < x2) element-wise. |
| `compare_chararrays`(a, b, cmp_op, rstrip) | Performs element-wise comparison of two string arrays using the comparison operator specified by *cmp_op*. |

`numpy.char.`**`equal`**`(`*x1*, *x2*`)`

Return (x1 == x2) element-wise.

Unlike *`numpy.equal`*, this comparison is performed by first stripping whitespace characters from the end of the string. This behavior is provided for backward-compatibility with numarray.

**Parameters**

**x1, x2**

[array_like of str or unicode] Input arrays of the same shape.

**Returns**

**out**

[ndarray or bool] Output array of bools, or a single bool if x1 and x2 are scalars.

**See also:**

*`not_equal`*, *`greater_equal`*, *`less_equal`*, *`greater`*, *`less`*

`numpy.char.`**`not_equal`**`(`*x1*, *x2*`)`

Return (x1 != x2) element-wise.

Unlike *`numpy.not_equal`*, this comparison is performed by first stripping whitespace characters from the end of the string. This behavior is provided for backward-compatibility with numarray.

**Parameters**

**x1, x2**

[array_like of str or unicode] Input arrays of the same shape.

**Returns**

**out**

[ndarray or bool] Output array of bools, or a single bool if x1 and x2 are scalars.

**See also:**

*`equal`*, *`greater_equal`*, *`less_equal`*, *`greater`*, *`less`*

`numpy.char.`**`greater_equal`**`(`*x1*, *x2*`)`

Return (x1 >= x2) element-wise.

Unlike *`numpy.greater_equal`*, this comparison is performed by first stripping whitespace characters from the end of the string. This behavior is provided for backward-compatibility with numarray.

**Parameters**

**x1, x2**

[array_like of str or unicode] Input arrays of the same shape.

**Returns**

**out**

[ndarray or bool] Output array of bools, or a single bool if x1 and x2 are scalars.

**See also:**

*equal*, *not_equal*, *less_equal*, *greater*, *less*

`numpy.char.`**`less_equal`**(*x1*, *x2*)

Return (x1 <= x2) element-wise.

Unlike *numpy.less_equal*, this comparison is performed by first stripping whitespace characters from the end of the string. This behavior is provided for backward-compatibility with numarray.

**Parameters**

**x1, x2**

[array_like of str or unicode] Input arrays of the same shape.

**Returns**

**out**

[ndarray or bool] Output array of bools, or a single bool if x1 and x2 are scalars.

**See also:**

*equal*, *not_equal*, *greater_equal*, *greater*, *less*

`numpy.char.`**`greater`**(*x1*, *x2*)

Return (x1 > x2) element-wise.

Unlike *numpy.greater*, this comparison is performed by first stripping whitespace characters from the end of the string. This behavior is provided for backward-compatibility with numarray.

**Parameters**

**x1, x2**

[array_like of str or unicode] Input arrays of the same shape.

**Returns**

**out**

[ndarray or bool] Output array of bools, or a single bool if x1 and x2 are scalars.

**See also:**

*equal*, *not_equal*, *greater_equal*, *less_equal*, *less*

`numpy.char.`**`less`**(*x1*, *x2*)

Return (x1 < x2) element-wise.

Unlike *numpy.greater*, this comparison is performed by first stripping whitespace characters from the end of the string. This behavior is provided for backward-compatibility with numarray.

**Parameters**

**x1, x2**

[array_like of str or unicode] Input arrays of the same shape.

**Returns**

**out**

[ndarray or bool] Output array of bools, or a single bool if x1 and x2 are scalars.

**See also:**

*equal*, *not_equal*, *greater_equal*, *less_equal*, *greater*

`numpy.char.compare_chararrays`(*a*, *b*, *cmp_op*, *rstrip*)

Performs element-wise comparison of two string arrays using the comparison operator specified by *cmp_op*.

**Parameters**

**a, b**

[array_like] Arrays to be compared.

**cmp_op**

[{"<", "<=", "==", ">=", ">", "!="}] Type of comparison.

**rstrip**

[Boolean] If True, the spaces at the end of Strings are removed before the comparison.

**Returns**

**out**

[ndarray] The output array of type Boolean with the same shape as a and b.

**Raises**

**ValueError**

If *cmp_op* is not valid.

**TypeError**

If at least one of *a* or *b* is a non-string array

### Examples

```
>>> a = np.array(["a", "b", "cde"])
>>> b = np.array(["a", "a", "dec"])
>>> np.compare_chararrays(a, b, ">", True)
array([False,  True, False])
```

## 4.4.3 String information

| | |
|---|---|
| `count`(a, sub[, start, end]) | Returns an array with the number of non-overlapping occurrences of substring *sub* in the range [*start*, *end*]. |
| `endswith`(a, suffix[, start, end]) | Returns a boolean array which is *True* where the string element in *a* ends with *suffix*, otherwise *False*. |
| `find`(a, sub[, start, end]) | For each element, return the lowest index in the string where substring *sub* is found. |

Continued on next page

Table 23 – continued from previous page

| | |
|---|---|
| `index`(a, sub[, start, end]) | Like `find`, but raises *ValueError* when the substring is not found. |
| `isalpha`(a) | Returns true for each element if all characters in the string are alphabetic and there is at least one character, false otherwise. |
| `isalnum`(a) | Returns true for each element if all characters in the string are alphanumeric and there is at least one character, false otherwise. |
| `isdecimal`(a) | For each element, return True if there are only decimal characters in the element. |
| `isdigit`(a) | Returns true for each element if all characters in the string are digits and there is at least one character, false otherwise. |
| `islower`(a) | Returns true for each element if all cased characters in the string are lowercase and there is at least one cased character, false otherwise. |
| `isnumeric`(a) | For each element, return True if there are only numeric characters in the element. |
| `isspace`(a) | Returns true for each element if there are only whitespace characters in the string and there is at least one character, false otherwise. |
| `istitle`(a) | Returns true for each element if the element is a titlecased string and there is at least one character, false otherwise. |
| `isupper`(a) | Returns true for each element if all cased characters in the string are uppercase and there is at least one character, false otherwise. |
| `rfind`(a, sub[, start, end]) | For each element in *a*, return the highest index in the string where substring *sub* is found, such that *sub* is contained within [*start*, *end*]. |
| `rindex`(a, sub[, start, end]) | Like `rfind`, but raises *ValueError* when the substring *sub* is not found. |
| `startswith`(a, prefix[, start, end]) | Returns a boolean array which is *True* where the string element in *a* starts with *prefix*, otherwise *False*. |
| `str_len`(a) | Return len(a) element-wise. |

`numpy.char.`**`count`**(*a*, *sub*, *start=0*, *end=None*)

Returns an array with the number of non-overlapping occurrences of substring *sub* in the range [*start*, *end*].

Calls *str.count* element-wise.

**Parameters**

**a**

[array_like of str or unicode]

**sub**

[str or unicode] The substring to search for.

**start, end**

[int, optional] Optional arguments *start* and *end* are interpreted as slice notation to specify the range in which to count.

**Returns**

**out**

[ndarray] Output array of ints.

**See also:**

`str.count`

### Examples

```
>>> c = np.array(['aAaAaA', '  aA  ', 'abBABba'])
>>> c
array(['aAaAaA', '  aA  ', 'abBABba'], dtype='<U7')
>>> np.char.count(c, 'A')
array([3, 1, 1])
>>> np.char.count(c, 'aA')
array([3, 1, 0])
>>> np.char.count(c, 'A', start=1, end=4)
array([2, 1, 1])
>>> np.char.count(c, 'A', start=1, end=3)
array([1, 0, 0])
```

`numpy.char.`**`endswith`**(*a*, *suffix*, *start=0*, *end=None*)

Returns a boolean array which is *True* where the string element in *a* ends with *suffix*, otherwise *False*.

Calls *str.endswith* element-wise.

**Parameters**

**a**

[array_like of str or unicode]

**suffix**

[str]

**start, end**

[int, optional] With optional *start*, test beginning at that position. With optional *end*, stop comparing at that position.

**Returns**

**out**

[ndarray] Outputs an array of bools.

**See also:**

`str.endswith`

### Examples

```
>>> s = np.array(['foo', 'bar'])
>>> s[0] = 'foo'
>>> s[1] = 'bar'
>>> s
array(['foo', 'bar'], dtype='<U3')
>>> np.char.endswith(s, 'ar')
array([False,  True])
>>> np.char.endswith(s, 'a', start=1, end=2)
array([False,  True])
```

`numpy.char.`**`find`**`(`*a*, *sub*, *start=0*, *end=None*`)`

For each element, return the lowest index in the string where substring *sub* is found.

Calls *str.find* element-wise.

For each element, return the lowest index in the string where substring *sub* is found, such that *sub* is contained in the range [*start*, *end*].

**Parameters**

**a**

[array_like of str or unicode]

**sub**

[str or unicode]

**start, end**

[int, optional] Optional arguments *start* and *end* are interpreted as in slice notation.

**Returns**

**out**

[ndarray or int] Output array of ints. Returns -1 if *sub* is not found.

**See also:**

`str.find`

`numpy.char.`**`index`**`(`*a*, *sub*, *start=0*, *end=None*`)`

Like `find`, but raises *ValueError* when the substring is not found.

Calls *str.index* element-wise.

**Parameters**

**a**

[array_like of str or unicode]

**sub**

[str or unicode]

**start, end**

[int, optional]

**Returns**

**out**

[ndarray] Output array of ints. Returns -1 if *sub* is not found.

**See also:**

*find*, `str.find`

`numpy.char.`**`isalpha`**`(`*a*`)`

Returns true for each element if all characters in the string are alphabetic and there is at least one character, false otherwise.

Calls *str.isalpha* element-wise.

For 8-bit strings, this method is locale-dependent.

**Parameters**

**a**

[array_like of str or unicode]

**Returns**

**out**

[ndarray] Output array of bools

**See also:**

`str.isalpha`

`numpy.char.`**`isalnum`**`(`*a*`)`

Returns true for each element if all characters in the string are alphanumeric and there is at least one character, false otherwise.

Calls *str.isalnum* element-wise.

For 8-bit strings, this method is locale-dependent.

**Parameters**

**a**

[array_like of str or unicode]

**Returns**

**out**

[ndarray] Output array of str or unicode, depending on input type

**See also:**

`str.isalnum`

`numpy.char.`**`isdecimal`**`(`*a*`)`

For each element, return True if there are only decimal characters in the element.

Calls *unicode.isdecimal* element-wise.

Decimal characters include digit characters, and all characters that can be used to form decimal-radix numbers, e.g. `U+0660, ARABIC-INDIC DIGIT ZERO`.

**Parameters**

**a**

[array_like, unicode] Input array.

**Returns**

**out**

[ndarray, bool] Array of booleans identical in shape to *a*.

**See also:**

`unicode.isdecimal`

`numpy.char.`**`isdigit`**`(`*a*`)`

Returns true for each element if all characters in the string are digits and there is at least one character, false otherwise.

Calls *str.isdigit* element-wise.

For 8-bit strings, this method is locale-dependent.

**Parameters**

**a**

[array_like of str or unicode]

**Returns**

**out**

[ndarray] Output array of bools

**See also:**

`str.isdigit`

`numpy.char.`**`islower`**`(`*a*`)`

Returns true for each element if all cased characters in the string are lowercase and there is at least one cased character, false otherwise.

Calls *str.islower* element-wise.

For 8-bit strings, this method is locale-dependent.

**Parameters**

**a**

[array_like of str or unicode]

**Returns**

**out**

[ndarray] Output array of bools

**See also:**

`str.islower`

`numpy.char.`**`isnumeric`**`(a)`

For each element, return True if there are only numeric characters in the element.

Calls *unicode.isnumeric* element-wise.

Numeric characters include digit characters, and all characters that have the Unicode numeric value property, e.g. `U+2155, VULGAR FRACTION ONE FIFTH`.

**Parameters**

**a**

[array_like, unicode] Input array.

**Returns**

**out**

[ndarray, bool] Array of booleans of same shape as *a*.

**See also:**

`unicode.isnumeric`

`numpy.char.`**`isspace`**`(a)`

Returns true for each element if there are only whitespace characters in the string and there is at least one character, false otherwise.

Calls *str.isspace* element-wise.

For 8-bit strings, this method is locale-dependent.

**Parameters**

**a**

[array_like of str or unicode]

**Returns**

**out**

[ndarray] Output array of bools

**See also:**

`str.isspace`

`numpy.char.`**`istitle`**`(a)`

Returns true for each element if the element is a titlecased string and there is at least one character, false otherwise.

Call *str.istitle* element-wise.

For 8-bit strings, this method is locale-dependent.

**Parameters**

**a**

[array_like of str or unicode]

**Returns**

**out**

[ndarray] Output array of bools

**See also:**

`str.istitle`

`numpy.char.`**`isupper`**`(`*a*`)`

Returns true for each element if all cased characters in the string are uppercase and there is at least one character, false otherwise.

Call *str.isupper* element-wise.

For 8-bit strings, this method is locale-dependent.

**Parameters**

**a**

[array_like of str or unicode]

**Returns**

**out**

[ndarray] Output array of bools

**See also:**

`str.isupper`

`numpy.char.`**`rfind`**`(`*a*, *sub*, *start=0*, *end=None*`)`

For each element in *a*, return the highest index in the string where substring *sub* is found, such that *sub* is contained within [*start*, *end*].

Calls *str.rfind* element-wise.

**Parameters**

**a**

[array-like of str or unicode]

**sub**

[str or unicode]

**start, end**

[int, optional] Optional arguments *start* and *end* are interpreted as in slice notation.

**Returns**

**out**

[ndarray] Output array of ints. Return -1 on failure.

**See also:**

`str.rfind`

`numpy.char.`**`rindex`**`(`*a*, *sub*, *start=0*, *end=None*`)`

Like `rfind`, but raises *ValueError* when the substring *sub* is not found.

Calls *str.rindex* element-wise.

**Parameters**

**a**

[array-like of str or unicode]

**sub**

[str or unicode]

**start, end**

[int, optional]

**Returns**

**out**

[ndarray] Output array of ints.

**See also:**

`rfind`, `str.rindex`

`numpy.char.`**`startswith`**`(`*a*, *prefix*, *start=0*, *end=None*`)`

Returns a boolean array which is *True* where the string element in *a* starts with *prefix*, otherwise *False*.

Calls *str.startswith* element-wise.

**Parameters**

**a**

[array_like of str or unicode]

**prefix**

[str]

**start, end**

[int, optional] With optional *start*, test beginning at that position. With optional *end*, stop comparing at that position.

**Returns**

**out**

[ndarray] Array of booleans

**See also:**

`str.startswith`

`numpy.char.`**`str_len`**`(`*a*`)`

Return len(a) element-wise.

**Parameters**

**a**

[array_like of str or unicode]

**Returns**

**out**

[ndarray] Output array of integers

**See also:**

`builtins.len`

### 4.4.4 Convenience class

| | |
|---|---|
| `array`(obj[, itemsize, copy, unicode, order]) | Create a `chararray`. |
| `asarray`(obj[, itemsize, unicode, order]) | Convert the input to a `chararray`, copying the data only if necessary. |
| `chararray`(shape[, itemsize, unicode, …]) | Provides a convenient view on arrays of string and unicode values. |

`numpy.char.`**`array`**(*obj*, *itemsize=None*, *copy=True*, *unicode=None*, *order=None*)

Create a `chararray`.

**Note:** This class is provided for numarray backward-compatibility. New code (not concerned with numarray compatibility) should use arrays of type `string_` or `unicode_` and use the free functions in `numpy.char` for fast vectorized string operations instead.

Versus a regular NumPy array of type *str* or *unicode*, this class adds the following functionality:

1) values automatically have whitespace removed from the end when indexed
2) comparison operators automatically remove whitespace from the end when comparing values
3) vectorized string operations are provided as methods (e.g. *str.endswith*) and infix operators (e.g. `+, *, %`)

**Parameters**

**obj**

[array of str or unicode-like]

**itemsize**

[int, optional] *itemsize* is the number of characters per scalar in the resulting array. If *itemsize* is None, and *obj* is an object array or a Python list, the *itemsize* will be automatically determined. If *itemsize* is provided and *obj* is of type str or unicode, then the *obj* string will be chunked into *itemsize* pieces.

**copy**

[bool, optional] If true (default), then the object is copied. Otherwise, a copy will only be made if __array__ returns a copy, if obj is a nested sequence, or if a copy is needed to satisfy any of the other requirements (*itemsize*, unicode, *order*, etc.).

**unicode**

[bool, optional] When true, the resulting `chararray` can contain Unicode characters, when false only 8-bit characters. If unicode is None and *obj* is one of the following:

- a `chararray`,
- an ndarray of type *str* or *unicode*
- a Python str or unicode object,

then the unicode setting of the output array will be automatically determined.

**order**

[{'C', 'F', 'A'}, optional] Specify the order of the array. If order is 'C' (default), then the array will be in C-contiguous order (last-index varies the fastest). If order is 'F', then the returned array will be in Fortran-contiguous order (first-index varies the fastest). If order is 'A', then the returned array may be in any order (either C-, Fortran-contiguous, or even discontiguous).

`numpy.char.`**`asarray`**(*obj*, *itemsize=None*, *unicode=None*, *order=None*)

Convert the input to a `chararray`, copying the data only if necessary.

Versus a regular NumPy array of type *str* or *unicode*, this class adds the following functionality:

1) values automatically have whitespace removed from the end when indexed
2) comparison operators automatically remove whitespace from the end when comparing values
3) vectorized string operations are provided as methods (e.g. *str.endswith*) and infix operators (e.g. `+`, `*`,"%")

**Parameters**

**obj**

[array of str or unicode-like]

**itemsize**

[int, optional] *itemsize* is the number of characters per scalar in the resulting array. If *itemsize* is None, and *obj* is an object array or a Python list, the *itemsize* will be automatically determined. If *itemsize* is provided and *obj* is of type str or unicode, then the *obj* string will be chunked into *itemsize* pieces.

**unicode**

[bool, optional] When true, the resulting `chararray` can contain Unicode characters, when false only 8-bit characters. If unicode is None and *obj* is one of the following:

- a `chararray`,
- an ndarray of type *str* or 'unicode'
- a Python str or unicode object,

then the unicode setting of the output array will be automatically determined.

**order**

[{'C', 'F'}, optional] Specify the order of the array. If order is 'C' (default), then the array will be in C-contiguous order (last-index varies the fastest). If order is 'F', then the returned array will be in Fortran-contiguous order (first-index varies the fastest).

**class** `numpy.char.`**`chararray`**(*shape*, *itemsize=1*, *unicode=False*, *buffer=None*, *offset=0*, *strides=None*, *order=None*)

Provides a convenient view on arrays of string and unicode values.

---

**Note:** The `chararray` class exists for backwards compatibility with Numarray, it is not recommended for new development. Starting from numpy 1.4, if one needs arrays of strings, it is recommended to use arrays of `dtype` `object_`, `string_` or `unicode_`, and use the free functions in the `numpy.char` module for fast vectorized string operations.

---

Versus a regular NumPy array of type *str* or *unicode*, this class adds the following functionality:

1) values automatically have whitespace removed from the end when indexed
2) comparison operators automatically remove whitespace from the end when comparing values
3) vectorized string operations are provided as methods (e.g. `endswith`) and infix operators (e.g. `"+", "*", "%"`)

chararrays should be created using `numpy.char.array` or `numpy.char.asarray`, rather than this constructor directly.

This constructor creates the array, using *buffer* (with *offset* and `strides`) if it is not `None`. If *buffer* is `None`, then constructs a new array with `strides` in “C order”, unless both `len(shape) >= 2` and `order='F'`, in which case `strides` is in “Fortran order”.

**Parameters**

**shape**

[tuple] Shape of the array.

**itemsize**

[int, optional] Length of each array element, in number of characters. Default is 1.

**unicode**

[bool, optional] Are the array elements of type unicode (True) or string (False). Default is False.

**buffer**

[object exposing the buffer interface or str, optional] Memory address of the start of the array data. Default is None, in which case a new array is created.

**offset**

[int, optional] Fixed stride displacement from the beginning of an axis? Default is 0. Needs to be >=0.

**strides**

[array_like of ints, optional] Strides for the array (see `ndarray.strides` for full description). Default is None.

**order**

[{‘C’, ‘F’}, optional] The order in which the array data is stored in memory: ‘C’ -> “row major” order (the default), ‘F’ -> “column major” (Fortran) order.

### Examples

```
>>> charar = np.chararray((3, 3))
>>> charar[:] = 'a'
>>> charar
chararray([[b'a', b'a', b'a'],
           [b'a', b'a', b'a'],
           [b'a', b'a', b'a']], dtype='|S1')
```

```
>>> charar = np.chararray(charar.shape, itemsize=5)
>>> charar[:] = 'abc'
>>> charar
chararray([[b'abc', b'abc', b'abc'],
           [b'abc', b'abc', b'abc'],
           [b'abc', b'abc', b'abc']], dtype='|S5')
```

**Attributes**

**`T`**

The transposed array.

**`base`**

Base object if memory is from some other object.

**`ctypes`**

An object to simplify the interaction of the array with the ctypes module.

**`data`**

Python buffer object pointing to the start of the array’s data.

**`dtype`**

Data-type of the array’s elements.

**`flags`**

Information about the memory layout of the array.

**`flat`**

A 1-D iterator over the array.

**`imag`**

The imaginary part of the array.

**`itemsize`**

Length of one array element in bytes.

**`nbytes`**

Total bytes consumed by the elements of the array.

**`ndim`**

Number of array dimensions.

**`real`**

The real part of the array.

**shape**

Tuple of array dimensions.

**size**

Number of elements in the array.

**strides**

Tuple of bytes to step in each dimension when traversing an array.

## Methods

| | |
|---|---|
| *astype*(dtype[, order, casting, subok, copy]) | Copy of the array, cast to a specified type. |
| *argsort*([axis, kind, order]) | Returns the indices that would sort this array. |
| *copy*([order]) | Return a copy of the array. |
| *count*(self, sub[, start, end]) | Returns an array with the number of non-overlapping occurrences of substring *sub* in the range [*start*, *end*]. |
| *decode*(self[, encoding, errors]) | Calls *str.decode* element-wise. |
| *dump*(file) | Dump a pickle of the array to the specified file. |
| *dumps*() | Returns the pickle of the array as a string. |
| *encode*(self[, encoding, errors]) | Calls *str.encode* element-wise. |
| *endswith*(self, suffix[, start, end]) | Returns a boolean array which is *True* where the string element in *self* ends with *suffix*, otherwise *False*. |
| *expandtabs*(self[, tabsize]) | Return a copy of each string element where all tab characters are replaced by one or more spaces. |
| *fill*(value) | Fill the array with a scalar value. |
| *find*(self, sub[, start, end]) | For each element, return the lowest index in the string where substring *sub* is found. |
| *flatten*([order]) | Return a copy of the array collapsed into one dimension. |
| *getfield*(dtype[, offset]) | Returns a field of the given array as a certain type. |
| *index*(self, sub[, start, end]) | Like *find*, but raises *ValueError* when the substring is not found. |
| *isalnum*(self) | Returns true for each element if all characters in the string are alphanumeric and there is at least one character, false otherwise. |
| *isalpha*(self) | Returns true for each element if all characters in the string are alphabetic and there is at least one character, false otherwise. |
| *isdecimal*(self) | For each element in *self*, return True if there are only decimal characters in the element. |
| *isdigit*(self) | Returns true for each element if all characters in the string are digits and there is at least one character, false otherwise. |
| *islower*(self) | Returns true for each element if all cased characters in the string are lowercase and there is at least one cased character, false otherwise. |
| *isnumeric*(self) | For each element in *self*, return True if there are only numeric characters in the element. |

Continued on next page

Table 25 – continued from previous page

| | |
|---|---|
| `isspace`(self) | Returns true for each element if there are only whitespace characters in the string and there is at least one character, false otherwise. |
| `istitle`(self) | Returns true for each element if the element is a titlecased string and there is at least one character, false otherwise. |
| `isupper`(self) | Returns true for each element if all cased characters in the string are uppercase and there is at least one character, false otherwise. |
| `item`(*args) | Copy an element of an array to a standard Python scalar and return it. |
| `join`(self, seq) | Return a string which is the concatenation of the strings in the sequence *seq*. |
| `ljust`(self, width[, fillchar]) | Return an array with the elements of *self* left-justified in a string of length *width*. |
| `lower`(self) | Return an array with the elements of *self* converted to lowercase. |
| `lstrip`(self[, chars]) | For each element in *self*, return a copy with the leading characters removed. |
| `nonzero`() | Return the indices of the elements that are non-zero. |
| `put`(indices, values[, mode]) | Set `a.flat[n] = values[n]` for all *n* in indices. |
| `ravel`([order]) | Return a flattened array. |
| `repeat`(repeats[, axis]) | Repeat elements of an array. |
| `replace`(self, old, new[, count]) | For each element in *self*, return a copy of the string with all occurrences of substring *old* replaced by *new*. |
| `reshape`(shape[, order]) | Returns an array containing the same data with a new shape. |
| `resize`(new_shape[, refcheck]) | Change shape and size of array in-place. |
| `rfind`(self, sub[, start, end]) | For each element in *self*, return the highest index in the string where substring *sub* is found, such that *sub* is contained within [*start*, *end*]. |
| `rindex`(self, sub[, start, end]) | Like `rfind`, but raises *ValueError* when the substring *sub* is not found. |
| `rjust`(self, width[, fillchar]) | Return an array with the elements of *self* right-justified in a string of length *width*. |
| `rsplit`(self[, sep, maxsplit]) | For each element in *self*, return a list of the words in the string, using *sep* as the delimiter string. |
| `rstrip`(self[, chars]) | For each element in *self*, return a copy with the trailing characters removed. |
| `searchsorted`(v[, side, sorter]) | Find indices where elements of v should be inserted in a to maintain order. |
| `setfield`(val, dtype[, offset]) | Put a value into a specified place in a field defined by a data-type. |
| `setflags`([write, align, uic]) | Set array flags WRITEABLE, ALIGNED, (WRITEBACKIFCOPY and UPDATEIFCOPY), respectively. |
| `sort`([axis, kind, order]) | Sort an array in-place. |
| `split`(self[, sep, maxsplit]) | For each element in *self*, return a list of the words in the string, using *sep* as the delimiter string. |

Continued on next page

Table 25 – continued from previous page

| | |
|---|---|
| `splitlines`(self[, keepends]) | For each element in *self*, return a list of the lines in the element, breaking at line boundaries. |
| `squeeze`([axis]) | Remove single-dimensional entries from the shape of *a*. |
| `startswith`(self, prefix[, start, end]) | Returns a boolean array which is *True* where the string element in *self* starts with *prefix*, otherwise *False*. |
| `strip`(self[, chars]) | For each element in *self*, return a copy with the leading and trailing characters removed. |
| `swapaxes`(axis1, axis2) | Return a view of the array with *axis1* and *axis2* interchanged. |
| `swapcase`(self) | For each element in *self*, return a copy of the string with uppercase characters converted to lowercase and vice versa. |
| `take`(indices[, axis, out, mode]) | Return an array formed from the elements of *a* at the given indices. |
| `title`(self) | For each element in *self*, return a titlecased version of the string: words start with uppercase characters, all remaining cased characters are lowercase. |
| `tofile`(fid[, sep, format]) | Write array to a file as text or binary (default). |
| `tolist`() | Return the array as an `a.ndim`-levels deep nested list of Python scalars. |
| `tostring`([order]) | A compatibility alias for `tobytes`, with exactly the same behavior. |
| `translate`(self, table[, deletechars]) | For each element in *self*, return a copy of the string where all characters occurring in the optional argument *deletechars* are removed, and the remaining characters have been mapped through the given translation table. |
| `transpose`(*axes) | Returns a view of the array with axes transposed. |
| `upper`(self) | Return an array with the elements of *self* converted to uppercase. |
| `view`([dtype][, type]) | New view of array with the same data. |
| `zfill`(self, width) | Return the numeric string left-filled with zeros in a string of length *width*. |

method

`chararray.`**`astype`**`(`*dtype*, *order='K'*, *casting='unsafe'*, *subok=True*, *copy=True*`)`

Copy of the array, cast to a specified type.

**Parameters**

**dtype**

[str or dtype] Typecode or data-type to which the array is cast.

**order**

[{'C', 'F', 'A', 'K'}, optional] Controls the memory layout order of the result. 'C' means C order, 'F' means Fortran order, 'A' means 'F' order if all the arrays are Fortran contiguous, 'C' order otherwise, and 'K' means as close to the order the array elements appear in memory as possible. Default is 'K'.

**casting**

[{'no', 'equiv', 'safe', 'same_kind', 'unsafe'}, optional] Controls what kind of data casting may occur. Defaults to 'unsafe' for backwards compatibility.

- 'no' means the data types should not be cast at all.
- 'equiv' means only byte-order changes are allowed.
- 'safe' means only casts which can preserve values are allowed.
- 'same_kind' means only safe casts or casts within a kind, like float64 to float32, are allowed.
- 'unsafe' means any data conversions may be done.

**subok**

[bool, optional] If True, then sub-classes will be passed-through (default), otherwise the returned array will be forced to be a base-class array.

**copy**

[bool, optional] By default, astype always returns a newly allocated array. If this is set to false, and the `dtype`, *order*, and *subok* requirements are satisfied, the input array is returned instead of a copy.

**Returns**

**arr_t**

[ndarray] Unless `copy` is False and the other conditions for returning the input array are satisfied (see description for `copy` input parameter), *arr_t* is a new array of the same shape as the input array, with dtype, order given by `dtype`, *order*.

**Raises**

**ComplexWarning**

When casting from complex to float or int. To avoid this, one should use `a.real.astype(t)`.

## Notes

Changed in version 1.17.0: Casting between a simple data type and a structured one is possible only for "unsafe" casting. Casting to multiple fields is allowed, but casting from multiple fields is not.

Changed in version 1.9.0: Casting from numeric to string types in 'safe' casting mode requires that the string dtype length is long enough to store the max integer/float value converted.

## Examples

```
>>> x = np.array([1, 2, 2.5])
>>> x
array([1. ,  2. ,  2.5])
```

```
>>> x.astype(int)
array([1, 2, 2])
```

method

`chararray.`**`argsort`**(*axis=-1*, *kind=None*, *order=None*)

Returns the indices that would sort this array.

Refer to `numpy.argsort` for full documentation.

**See also:**

**`numpy.argsort`**

equivalent function

method

`chararray.`**`copy`**(*order='C'*)

Return a copy of the array.

**Parameters**

**order**

[{'C', 'F', 'A', 'K'}, optional] Controls the memory layout of the copy. 'C' means C-order, 'F' means F-order, 'A' means 'F' if *a* is Fortran contiguous, 'C' otherwise. 'K' means match the layout of *a* as closely as possible. (Note that this function and `numpy.copy` are very similar, but have different default values for their order= arguments.)

**See also:**

`numpy.copy`, `numpy.copyto`

### Examples

```
>>> x = np.array([[1,2,3],[4,5,6]], order='F')
```

```
>>> y = x.copy()
```

```
>>> x.fill(0)
```

```
>>> x
array([[0, 0, 0],
       [0, 0, 0]])
```

```
>>> y
array([[1, 2, 3],
       [4, 5, 6]])
```

```
>>> y.flags['C_CONTIGUOUS']
True
```

method

`chararray.`**`count`**(*self*, *sub*, *start=0*, *end=None*)

Returns an array with the number of non-overlapping occurrences of substring *sub* in the range [*start*, *end*].

**See also:**

`char.count`

method

`chararray.`**`decode`**`(`*self*, *encoding=None*, *errors=None*`)`
Calls *str.decode* element-wise.

**See also:**

`char.decode`

method

`chararray.`**`dump`**`(`*file*`)`
Dump a pickle of the array to the specified file. The array can be read back with pickle.load or numpy.load.

**Parameters**

**file**

[str or Path] A string naming the dump file.

Changed in version 1.17.0: `pathlib.Path` objects are now accepted.

method

`chararray.`**`dumps`**`()`
Returns the pickle of the array as a string. pickle.loads or numpy.loads will convert the string back to an array.

**Parameters**

**None**

method

`chararray.`**`encode`**`(`*self*, *encoding=None*, *errors=None*`)`
Calls *str.encode* element-wise.

**See also:**

`char.encode`

method

`chararray.`**`endswith`**`(`*self*, *suffix*, *start=0*, *end=None*`)`
Returns a boolean array which is *True* where the string element in *self* ends with *suffix*, otherwise *False*.

**See also:**

`char.endswith`

method

`chararray.`**`expandtabs`**`(`*self*, *tabsize=8*`)`
Return a copy of each string element where all tab characters are replaced by one or more spaces.

**See also:**

`char.expandtabs`

method

`chararray.`**`fill`**`(`*value*`)`
Fill the array with a scalar value.

**Parameters**

**value**

[scalar] All elements of *a* will be assigned this value.

### Examples

```
>>> a = np.array([1, 2])
>>> a.fill(0)
>>> a
array([0, 0])
>>> a = np.empty(2)
>>> a.fill(1)
>>> a
array([1.,  1.])
```

method

`chararray.`**`find`**(*self*, *sub*, *start=0*, *end=None*)

For each element, return the lowest index in the string where substring *sub* is found.

**See also:**

`char.find`

method

`chararray.`**`flatten`**(*order='C'*)

Return a copy of the array collapsed into one dimension.

**Parameters**

**order**

[{'C', 'F', 'A', 'K'}, optional] 'C' means to flatten in row-major (C-style) order. 'F' means to flatten in column-major (Fortran- style) order. 'A' means to flatten in column-major order if *a* is Fortran *contiguous* in memory, row-major order otherwise. 'K' means to flatten *a* in the order the elements occur in memory. The default is 'C'.

**Returns**

**y**

[ndarray] A copy of the input array, flattened to one dimension.

**See also:**

***`ravel`***

Return a flattened array.

**`flat`**

A 1-D flat iterator over the array.

### Examples

```
>>> a = np.array([[1,2], [3,4]])
>>> a.flatten()
array([1, 2, 3, 4])
>>> a.flatten('F')
array([1, 3, 2, 4])
```

method

`chararray.`**`getfield`**`(`*dtype*, *offset=0*`)`

Returns a field of the given array as a certain type.

A field is a view of the array data with a given data-type. The values in the view are determined by the given type and the offset into the current array in bytes. The offset needs to be such that the view dtype fits in the array dtype; for example an array of dtype complex128 has 16-byte elements. If taking a view with a 32-bit integer (4 bytes), the offset needs to be between 0 and 12 bytes.

**Parameters**

**dtype**

[str or dtype] The data type of the view. The dtype size of the view can not be larger than that of the array itself.

**offset**

[int] Number of bytes to skip before beginning the element view.

### Examples

```
>>> x = np.diag([1.+1.j]*2)
>>> x[1, 1] = 2 + 4.j
>>> x
array([[1.+1.j,  0.+0.j],
       [0.+0.j,  2.+4.j]])
>>> x.getfield(np.float64)
array([[1.,  0.],
       [0.,  2.]])
```

By choosing an offset of 8 bytes we can select the complex part of the array for our view:

```
>>> x.getfield(np.float64, offset=8)
array([[1.,  0.],
       [0.,  4.]])
```

method

`chararray.`**`index`**`(`*self*, *sub*, *start=0*, *end=None*`)`

Like *`find`*, but raises *ValueError* when the substring is not found.

**See also:**

`char.index`

method

`chararray.`**`isalnum`**`(`*self*`)`

Returns true for each element if all characters in the string are alphanumeric and there is at least one character, false otherwise.

**See also:**

`char.isalnum`

method

`chararray.`**`isalpha`**`(`*self*`)`

Returns true for each element if all characters in the string are alphabetic and there is at least one character, false otherwise.

**See also:**

`char.isalpha`

method

`chararray.`**`isdecimal`**`(`*self*`)`

For each element in *self*, return True if there are only decimal characters in the element.

**See also:**

`char.isdecimal`

method

`chararray.`**`isdigit`**`(`*self*`)`

Returns true for each element if all characters in the string are digits and there is at least one character, false otherwise.

**See also:**

`char.isdigit`

method

`chararray.`**`islower`**`(`*self*`)`

Returns true for each element if all cased characters in the string are lowercase and there is at least one cased character, false otherwise.

**See also:**

`char.islower`

method

`chararray.`**`isnumeric`**`(`*self*`)`

For each element in *self*, return True if there are only numeric characters in the element.

**See also:**

`char.isnumeric`

method

`chararray.`**`isspace`**`(`*self*`)`

Returns true for each element if there are only whitespace characters in the string and there is at least one character, false otherwise.

**See also:**

`char.isspace`

method

`chararray.`**`istitle`**`(`*self*`)`

Returns true for each element if the element is a titlecased string and there is at least one character, false otherwise.

**See also:**

`char.istitle`

method

`chararray.`**`isupper`**`(`*self*`)`

Returns true for each element if all cased characters in the string are uppercase and there is at least one character, false otherwise.

**See also:**

`char.isupper`

method

`chararray.`**`item`**`(`**args*`)`

Copy an element of an array to a standard Python scalar and return it.

**Parameters**

***args**

[Arguments (variable number and type)]

- none: in this case, the method only works for arrays with one element (*a.size == 1*), which element is copied into a standard Python scalar object and returned.
- int_type: this argument is interpreted as a flat index into the array, specifying which element to copy and return.
- tuple of int_types: functions as does a single int_type argument, except that the argument is interpreted as an nd-index into the array.

**Returns**

**z**

[Standard Python scalar object] A copy of the specified element of the array as a suitable Python scalar

## Notes

When the data type of *a* is longdouble or clongdouble, item() returns a scalar array object because there is no available Python scalar that would not lose information. Void arrays return a buffer object for item(), unless fields are defined, in which case a tuple is returned.

*`item`* is very similar to a[args], except, instead of an array scalar, a standard Python scalar is returned. This can be useful for speeding up access to elements of the array and doing arithmetic on elements of the array using Python's optimized math.

### Examples

```
>>> np.random.seed(123)
>>> x = np.random.randint(9, size=(3, 3))
>>> x
array([[2, 2, 6],
       [1, 3, 6],
       [1, 0, 1]])
>>> x.item(3)
1
>>> x.item(7)
0
>>> x.item((0, 1))
2
>>> x.item((2, 2))
1
```

method

`chararray.`**`join`**`(`*self*, *seq*`)`

Return a string which is the concatenation of the strings in the sequence *seq*.

**See also:**

`char.join`

method

`chararray.`**`ljust`**`(`*self*, *width*, *fillchar=' '*`)`

Return an array with the elements of *self* left-justified in a string of length *width*.

**See also:**

`char.ljust`

method

`chararray.`**`lower`**`(`*self*`)`

Return an array with the elements of *self* converted to lowercase.

**See also:**

`char.lower`

method

`chararray.`**`lstrip`**`(`*self*, *chars=None*`)`

For each element in *self*, return a copy with the leading characters removed.

**See also:**

`char.lstrip`

method

`chararray.`**`nonzero`**`()`

Return the indices of the elements that are non-zero.

Refer to `numpy.nonzero` for full documentation.

**See also:**

**`numpy.nonzero`**

equivalent function

method

`chararray.`**`put`**`(`*indices*, *values*, *mode='raise'*`)`

Set `a.flat[n] = values[n]` for all *n* in indices.

Refer to `numpy.put` for full documentation.

**See also:**

**`numpy.put`**

equivalent function

method

`chararray.`**`ravel`**`(`[*order*]`)`

Return a flattened array.

Refer to `numpy.ravel` for full documentation.

**See also:**

**`numpy.ravel`**

equivalent function

**`ndarray.flat`**

a flat iterator on the array.

method

`chararray.`**`repeat`**`(`*repeats*, *axis=None*`)`

Repeat elements of an array.

Refer to `numpy.repeat` for full documentation.

**See also:**

**`numpy.repeat`**

equivalent function

method

`chararray.`**`replace`**`(`*self*, *old*, *new*, *count=None*`)`

For each element in *self*, return a copy of the string with all occurrences of substring *old* replaced by *new*.

**See also:**

`char.replace`

method

`chararray.`**`reshape`**`(`*shape*, *order='C'*`)`

Returns an array containing the same data with a new shape.

Refer to `numpy.reshape` for full documentation.

**See also:**

**`numpy.reshape`**

equivalent function

### Notes

Unlike the free function *`numpy.reshape`*, this method on `ndarray` allows the elements of the shape parameter to be passed in as separate arguments. For example, `a.reshape(10, 11)` is equivalent to `a.reshape((10, 11))`.

method

`chararray.`**`resize`**`(`*new_shape*, *refcheck=True*`)`

Change shape and size of array in-place.

**Parameters**

**new_shape**

[tuple of ints, or *n* ints] Shape of resized array.

**refcheck**

[bool, optional] If False, reference count will not be checked. Default is True.

**Returns**

**None**

**Raises**

**ValueError**

If *a* does not own its own data or references or views to it exist, and the data memory must be changed. PyPy only: will always raise if the data memory must be changed, since there is no reliable way to determine if references or views to it exist.

**SystemError**

If the *order* keyword argument is specified. This behaviour is a bug in NumPy.

**See also:**

***`resize`***

Return a new array with the specified shape.

### Notes

This reallocates space for the data area if necessary.

Only contiguous arrays (data elements consecutive in memory) can be resized.

The purpose of the reference count check is to make sure you do not use this array as a buffer for another Python object and then reallocate the memory. However, reference counts can increase in other ways so if you are sure that you have not shared the memory for this array with another Python object, then you may safely set *refcheck* to False.

### Examples

Shrinking an array: array is flattened (in the order that the data are stored in memory), resized, and reshaped:

```
>>> a = np.array([[0, 1], [2, 3]], order='C')
>>> a.resize((2, 1))
>>> a
array([[0],
       [1]])
```

```
>>> a = np.array([[0, 1], [2, 3]], order='F')
>>> a.resize((2, 1))
>>> a
array([[0],
       [2]])
```

Enlarging an array: as above, but missing entries are filled with zeros:

```
>>> b = np.array([[0, 1], [2, 3]])
>>> b.resize(2, 3) # new_shape parameter doesn't have to be a tuple
>>> b
array([[0, 1, 2],
       [3, 0, 0]])
```

Referencing an array prevents resizing…

```
>>> c = a
>>> a.resize((1, 1))
Traceback (most recent call last):
...
ValueError: cannot resize an array that references or is referenced ...
```

Unless *refcheck* is False:

```
>>> a.resize((1, 1), refcheck=False)
>>> a
array([[0]])
>>> c
array([[0]])
```

method

`chararray.`**`rfind`**(*self*, *sub*, *start=0*, *end=None*)

For each element in *self*, return the highest index in the string where substring *sub* is found, such that *sub* is contained within [*start*, *end*].

**See also:**

`char.rfind`

method

`chararray.`**`rindex`**(*self*, *sub*, *start=0*, *end=None*)

Like `rfind`, but raises *ValueError* when the substring *sub* is not found.

**See also:**

`char.rindex`

method

`chararray.`**`rjust`**(*self*, *width*, *fillchar=' '*)

Return an array with the elements of *self* right-justified in a string of length *width*.

**See also:**

`char.rjust`

method

`chararray.`**`rsplit`**(*self*, *sep=None*, *maxsplit=None*)

For each element in *self*, return a list of the words in the string, using *sep* as the delimiter string.

**See also:**

`char.rsplit`

method

`chararray.`**`rstrip`**(*self*, *chars=None*)

For each element in *self*, return a copy with the trailing characters removed.

**See also:**

`char.rstrip`

method

`chararray.`**`searchsorted`**(*v*, *side='left'*, *sorter=None*)

Find indices where elements of v should be inserted in a to maintain order.

For full documentation, see `numpy.searchsorted`

**See also:**

**`numpy.searchsorted`**

equivalent function

method

`chararray.`**`setfield`**(*val*, *dtype*, *offset=0*)

Put a value into a specified place in a field defined by a data-type.

Place *val* into *a*'s field defined by `dtype` and beginning *offset* bytes into the field.

**Parameters**

**val**

[object] Value to be placed in field.

**dtype**

[dtype object] Data-type of the field in which to place *val*.

**offset**

[int, optional] The number of bytes into the field at which to place *val*.

**Returns**

**None**

**See also:**

`getfield`

## Examples

```
>>> x = np.eye(3)
>>> x.getfield(np.float64)
array([[1.,  0.,  0.],
       [0.,  1.,  0.],
       [0.,  0.,  1.]])
>>> x.setfield(3, np.int32)
>>> x.getfield(np.int32)
array([[3, 3, 3],
       [3, 3, 3],
       [3, 3, 3]], dtype=int32)
>>> x
array([[1.0e+000, 1.5e-323, 1.5e-323],
       [1.5e-323, 1.0e+000, 1.5e-323],
       [1.5e-323, 1.5e-323, 1.0e+000]])
>>> x.setfield(np.eye(3), np.int32)
>>> x
array([[1.,  0.,  0.],
       [0.,  1.,  0.],
       [0.,  0.,  1.]])
```

method

`chararray.`**`setflags`**`(`*write=None*, *align=None*, *uic=None*`)`

Set array flags WRITEABLE, ALIGNED, (WRITEBACKIFCOPY and UPDATEIFCOPY), respectively.

These Boolean-valued flags affect how numpy interprets the memory area used by *a* (see Notes below). The ALIGNED flag can only be set to True if the data is actually aligned according to the type. The WRITEBACKIFCOPY and (deprecated) UPDATEIFCOPY flags can never be set to True. The flag WRITEABLE can only be set to True if the array owns its own memory, or the ultimate owner of the memory exposes a writeable buffer interface, or is a string. (The exception for string is made so that unpickling can be done without copying memory.)

**Parameters**

**write**

[bool, optional] Describes whether or not *a* can be written to.

**align**

[bool, optional] Describes whether or not *a* is aligned properly for its type.

**uic**

[bool, optional] Describes whether or not *a* is a copy of another "base" array.

## Notes

Array flags provide information about how the memory area used for the array is to be interpreted. There are 7 Boolean flags in use, only four of which can be changed by the user: WRITEBACKIFCOPY, UPDATEIFCOPY, WRITEABLE, and ALIGNED.

WRITEABLE (W) the data area can be written to;

ALIGNED (A) the data and strides are aligned appropriately for the hardware (as determined by the compiler);

UPDATEIFCOPY (U) (deprecated), replaced by WRITEBACKIFCOPY;

WRITEBACKIFCOPY (X) this array is a copy of some other array (referenced by .base). When the C-API function PyArray_ResolveWritebackIfCopy is called, the base array will be updated with the contents of this array.

All flags can be accessed using the single (upper case) letter as well as the full name.

### Examples

```
>>> y = np.array([[3, 1, 7],
...               [2, 0, 0],
...               [8, 5, 9]])
>>> y
array([[3, 1, 7],
       [2, 0, 0],
       [8, 5, 9]])
>>> y.flags
  C_CONTIGUOUS : True
  F_CONTIGUOUS : False
  OWNDATA : True
  WRITEABLE : True
  ALIGNED : True
  WRITEBACKIFCOPY : False
  UPDATEIFCOPY : False
>>> y.setflags(write=0, align=0)
>>> y.flags
  C_CONTIGUOUS : True
  F_CONTIGUOUS : False
  OWNDATA : True
  WRITEABLE : False
  ALIGNED : False
  WRITEBACKIFCOPY : False
  UPDATEIFCOPY : False
>>> y.setflags(uic=1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: cannot set WRITEBACKIFCOPY flag to True
```

method

`chararray.`**`sort`**(*axis=-1*, *kind=None*, *order=None*)

Sort an array in-place. Refer to `numpy.sort` for full documentation.

**Parameters**

**axis**

[int, optional] Axis along which to sort. Default is -1, which means sort along the last axis.

**kind**

[{'quicksort', 'mergesort', 'heapsort', 'stable'}, optional] Sorting algorithm. The default is 'quicksort'. Note that both 'stable' and 'mergesort' use timsort under the covers and, in general, the actual implementation will vary with datatype. The 'mergesort' option is retained for backwards compatibility.

Changed in version 1.15.0.: The 'stable' option was added.

**order**

[str or list of str, optional] When *a* is an array with fields defined, this argument specifies which fields to compare first, second, etc. A single field can be specified as a string, and not all fields need be specified, but unspecified fields will still be used, in the order in which they come up in the dtype, to break ties.

**See also:**

***`numpy.sort`***

Return a sorted copy of an array.

***`numpy.argsort`***

Indirect sort.

***`numpy.lexsort`***

Indirect stable sort on multiple keys.

***`numpy.searchsorted`***

Find elements in sorted array.

***`numpy.partition`***

Partial sort.

### Notes

See `numpy.sort` for notes on the different sorting algorithms.

### Examples

```
>>> a = np.array([[1,4], [3,1]])
>>> a.sort(axis=1)
>>> a
array([[1, 4],
       [1, 3]])
>>> a.sort(axis=0)
>>> a
array([[1, 3],
       [1, 4]])
```

Use the *order* keyword to specify a field to use when sorting a structured array:

```
>>> a = np.array([('a', 2), ('c', 1)], dtype=[('x', 'S1'), ('y', int)])
>>> a.sort(order='y')
>>> a
array([(b'c', 1), (b'a', 2)],
      dtype=[('x', 'S1'), ('y', '<i8')])
```

method

`chararray.`**`split`**(*self*, *sep=None*, *maxsplit=None*)

For each element in *self*, return a list of the words in the string, using *sep* as the delimiter string.

**See also:**

`char.split`

method

`chararray.`**`splitlines`**`(`*self*, *keepends=None*`)`
For each element in *self*, return a list of the lines in the element, breaking at line boundaries.

**See also:**

`char.splitlines`

method

`chararray.`**`squeeze`**`(`*axis=None*`)`
Remove single-dimensional entries from the shape of *a*.

Refer to *`numpy.squeeze`* for full documentation.

**See also:**

***`numpy.squeeze`***
equivalent function

method

`chararray.`**`startswith`**`(`*self*, *prefix*, *start=0*, *end=None*`)`
Returns a boolean array which is *True* where the string element in *self* starts with *prefix*, otherwise *False*.

**See also:**

`char.startswith`

method

`chararray.`**`strip`**`(`*self*, *chars=None*`)`
For each element in *self*, return a copy with the leading and trailing characters removed.

**See also:**

`char.strip`

method

`chararray.`**`swapaxes`**`(`*axis1*, *axis2*`)`
Return a view of the array with *axis1* and *axis2* interchanged.

Refer to *`numpy.swapaxes`* for full documentation.

**See also:**

***`numpy.swapaxes`***
equivalent function

method

`chararray.`**`swapcase`**`(`*self*`)`
For each element in *self*, return a copy of the string with uppercase characters converted to lowercase and vice versa.

**See also:**

`char.swapcase`

method

`chararray`.**`take`**(*indices*, *axis=None*, *out=None*, *mode='raise'*)

Return an array formed from the elements of *a* at the given indices.

Refer to `numpy.take` for full documentation.

**See also:**

**`numpy.take`**

equivalent function

method

`chararray`.**`title`**(*self*)

For each element in *self*, return a titlecased version of the string: words start with uppercase characters, all remaining cased characters are lowercase.

**See also:**

`char.title`

method

`chararray`.**`tofile`**(*fid*, *sep=""*, *format="%s"*)

Write array to a file as text or binary (default).

Data is always written in 'C' order, independent of the order of *a*. The data produced by this method can be recovered using the function fromfile().

**Parameters**

**fid**

[file or str or Path] An open file object, or a string containing a filename.

Changed in version 1.17.0: `pathlib.Path` objects are now accepted.

**sep**

[str] Separator between array items for text output. If "" (empty), a binary file is written, equivalent to `file.write(a.tobytes())`.

**format**

[str] Format string for text file output. Each entry in the array is formatted to text by first converting it to the closest Python type, and then using "format" % item.

## Notes

This is a convenience function for quick storage of array data. Information on endianness and precision is lost, so this method is not a good choice for files intended to archive data or transport data between machines with different endianness. Some of these problems can be overcome by outputting the data as text files, at the expense of speed and file size.

When fid is a file object, array contents are directly written to the file, bypassing the file object's `write` method. As a result, tofile cannot be used with files objects supporting compression (e.g., GzipFile) or file-like objects that do not support `fileno()` (e.g., BytesIO).

method

`chararray.tolist()`

Return the array as an `a.ndim`-levels deep nested list of Python scalars.

Return a copy of the array data as a (nested) Python list. Data items are converted to the nearest compatible builtin Python type, via the *item* function.

If `a.ndim` is 0, then since the depth of the nested list is 0, it will not be a list at all, but a simple Python scalar.

**Parameters**

**none**

**Returns**

**y**

[object, or list of object, or list of list of object, or …] The possibly nested list of array elements.

## Notes

The array may be recreated via `a = np.array(a.tolist())`, although this may sometimes lose precision.

## Examples

For a 1D array, `a.tolist()` is almost the same as `list(a)`, except that `tolist` changes numpy scalars to Python scalars:

```
>>> a = np.uint32([1, 2])
>>> a_list = list(a)
>>> a_list
[1, 2]
>>> type(a_list[0])
<class 'numpy.uint32'>
>>> a_tolist = a.tolist()
>>> a_tolist
[1, 2]
>>> type(a_tolist[0])
<class 'int'>
```

Additionally, for a 2D array, `tolist` applies recursively:

```
>>> a = np.array([[1, 2], [3, 4]])
>>> list(a)
[array([1, 2]), array([3, 4])]
>>> a.tolist()
[[1, 2], [3, 4]]
```

The base case for this recursion is a 0D array:

```
>>> a = np.array(1)
>>> list(a)
Traceback (most recent call last):
  ...
```

(continues on next page)

(continued from previous page)

```
TypeError: iteration over a 0-d array
>>> a.tolist()
1
```

method

`chararray.`**`tostring`**`(`*order='C'*`)`

A compatibility alias for `tobytes`, with exactly the same behavior.

Despite its name, it returns *bytes* not *str*s.

Deprecated since version 1.19.0.

method

`chararray.`**`translate`**`(`*self*, *table*, *deletechars=None*`)`

For each element in *self*, return a copy of the string where all characters occurring in the optional argument *deletechars* are removed, and the remaining characters have been mapped through the given translation table.

**See also:**

`char.translate`

method

`chararray.`**`transpose`**`(`**axes*`)`

Returns a view of the array with axes transposed.

For a 1-D array this has no effect, as a transposed vector is simply the same vector. To convert a 1-D array into a 2D column vector, an additional dimension must be added. *np.atleast2d(a).T* achieves this, as does *a[:, np.newaxis]*. For a 2-D array, this is a standard matrix transpose. For an n-D array, if axes are given, their order indicates how the axes are permuted (see Examples). If axes are not provided and `a.shape = (i[0], i[1], ... i[n-2], i[n-1])`, then `a.transpose().shape = (i[n-1], i[n-2], ... i[1], i[0])`.

**Parameters**

**axes**

[None, tuple of ints, or *n* ints]

- None or no argument: reverses the order of the axes.
- tuple of ints: *i* in the *j*-th place in the tuple means *a*'s *i*-th axis becomes *a.transpose()*'s *j*-th axis.
- *n* ints: same as an n-tuple of the same ints (this form is intended simply as a "convenience" alternative to the tuple form)

**Returns**

**out**

[ndarray] View of *a*, with axes suitably permuted.

**See also:**

**`ndarray.T`**

Array property returning the array transposed.

**`ndarray.reshape`**

Give a new shape to an array without changing its data.

### Examples

```
>>> a = np.array([[1, 2], [3, 4]])
>>> a
array([[1, 2],
       [3, 4]])
>>> a.transpose()
array([[1, 3],
       [2, 4]])
>>> a.transpose((1, 0))
array([[1, 3],
       [2, 4]])
>>> a.transpose(1, 0)
array([[1, 3],
       [2, 4]])
```

method

`chararray.`**`upper`**`(`*self*`)`

Return an array with the elements of *self* converted to uppercase.

**See also:**

`char.upper`

method

`chararray.`**`view`**`(`*[dtype][, type]*`)`

New view of array with the same data.

---

**Note:** Passing None for `dtype` is different from omitting the parameter, since the former invokes `dtype(None)` which is an alias for `dtype('float_')`.

---

**Parameters**

**dtype**

[data-type or ndarray sub-class, optional] Data-type descriptor of the returned view, e.g., float32 or int16. Omitting it results in the view having the same data-type as *a*. This argument can also be specified as an ndarray sub-class, which then specifies the type of the returned object (this is equivalent to setting the `type` parameter).

**type**

[Python type, optional] Type of the returned view, e.g., ndarray or matrix. Again, omission of the parameter results in type preservation.

### Notes

`a.view()` is used two different ways:

`a.view(some_dtype)` or `a.view(dtype=some_dtype)` constructs a view of the array's memory with a different data-type. This can cause a reinterpretation of the bytes of memory.

`a.view(ndarray_subclass)` or `a.view(type=ndarray_subclass)` just returns an instance of *ndarray_subclass* that looks at the same array (same shape, dtype, etc.) This does not cause a reinterpretation of the memory.

For `a.view(some_dtype)`, if `some_dtype` has a different number of bytes per entry than the previous dtype (for example, converting a regular array to a structured array), then the behavior of the view cannot be predicted just from the superficial appearance of `a` (shown by `print(a)`). It also depends on exactly how `a` is stored in memory. Therefore if `a` is C-ordered versus fortran-ordered, versus defined as a slice or transpose, etc., the view may give different results.

### Examples

```
>>> x = np.array([(1, 2)], dtype=[('a', np.int8), ('b', np.int8)])
```

Viewing array data using a different type and dtype:

```
>>> y = x.view(dtype=np.int16, type=np.matrix)
>>> y
matrix([[513]], dtype=int16)
>>> print(type(y))
<class 'numpy.matrix'>
```

Creating a view on a structured array so it can be used in calculations

```
>>> x = np.array([(1, 2),(3,4)], dtype=[('a', np.int8), ('b', np.int8)])
>>> xv = x.view(dtype=np.int8).reshape(-1,2)
>>> xv
array([[1, 2],
       [3, 4]], dtype=int8)
>>> xv.mean(0)
array([2.,  3.])
```

Making changes to the view changes the underlying array

```
>>> xv[0,1] = 20
>>> x
array([(1, 20), (3,  4)], dtype=[('a', 'i1'), ('b', 'i1')])
```

Using a view to convert an array to a recarray:

```
>>> z = x.view(np.recarray)
>>> z.a
array([1, 3], dtype=int8)
```

Views share data:

```
>>> x[0] = (9, 10)
>>> z[0]
(9, 10)
```

Views that change the dtype size (bytes per entry) should normally be avoided on arrays defined by slices, transposes, fortran-ordering, etc.:

```
>>> x = np.array([[1,2,3],[4,5,6]], dtype=np.int16)
>>> y = x[:, 0:2]
>>> y
array([[1, 2],
       [4, 5]], dtype=int16)
>>> y.view(dtype=[('width', np.int16), ('length', np.int16)])
Traceback (most recent call last):
    ...
ValueError: To change to a dtype of a different size, the array must be C-
↪contiguous
>>> z = y.copy()
>>> z.view(dtype=[('width', np.int16), ('length', np.int16)])
array([[(1, 2)],
       [(4, 5)]], dtype=[('width', '<i2'), ('length', '<i2')])
```

method

`chararray.`**`zfill`**`(`*self*, *width*`)`

Return the numeric string left-filled with zeros in a string of length *width*.

**See also:**

`char.zfill`

## 4.5 C-Types Foreign Function Interface (`numpy.ctypeslib`)

`numpy.ctypeslib.`**`as_array`**`(`*obj*, *shape=None*`)`

Create a numpy array from a ctypes array or POINTER.

The numpy array shares the memory with the ctypes object.

The shape parameter must be given if converting from a ctypes POINTER. The shape parameter is ignored if converting from a ctypes array

`numpy.ctypeslib.`**`as_ctypes`**`(`*obj*`)`

Create and return a ctypes object from a numpy array. Actually anything that exposes the __array_interface__ is accepted.

`numpy.ctypeslib.`**`as_ctypes_type`**`(`*dtype*`)`

Convert a dtype into a ctypes type.

**Parameters**

**dtype**

[dtype] The dtype to convert

**Returns**

**ctype**

A ctype scalar, union, array, or struct

**Raises**

**NotImplementedError**

If the conversion is not possible

### Notes

This function does not losslessly round-trip in either direction.

`np.dtype(as_ctypes_type(dt))` will:

- insert padding fields
- reorder fields to be sorted by offset
- discard field titles

`as_ctypes_type(np.dtype(ctype))` will:

- discard the class names of `ctypes.Structure`s and `ctypes.Union`s
- convert single-element `ctypes.Union`s into single-element `ctypes.Structure`s
- insert padding fields

`numpy.ctypeslib.`**`ctypes_load_library`**`(`**args*, ***kwds*`)`

*`ctypes_load_library`* is deprecated, use *`load_library`* instead!

It is possible to load a library using »> lib = ctypes.cdll[<full_path_name>] # doctest: +SKIP

But there are cross-platform considerations, such as library file extensions, plus the fact Windows will just load the first library it finds with that name. NumPy supplies the load_library function as a convenience.

**Parameters**

**libname**

[str] Name of the library, which can have 'lib' as a prefix, but without an extension.

**loader_path**

[str] Where the library can be found.

**Returns**

**ctypes.cdll[libpath]**

[library object] A ctypes library object

**Raises**

**OSError**

If there is no library with the expected extension, or the library is defective and cannot be loaded.

`numpy.ctypeslib.`**`load_library`**`(`*libname*, *loader_path*`)`

It is possible to load a library using »> lib = ctypes.cdll[<full_path_name>] # doctest: +SKIP

But there are cross-platform considerations, such as library file extensions, plus the fact Windows will just load the first library it finds with that name. NumPy supplies the load_library function as a convenience.

**Parameters**

**libname**

[str] Name of the library, which can have 'lib' as a prefix, but without an extension.

**loader_path**

[str] Where the library can be found.

**Returns**

**ctypes.cdll[libpath]**

[library object] A ctypes library object

**Raises**

**OSError**

If there is no library with the expected extension, or the library is defective and cannot be loaded.

`numpy.ctypeslib.`**`ndpointer`**(*dtype=None*, *ndim=None*, *shape=None*, *flags=None*)

Array-checking restype/argtypes.

An ndpointer instance is used to describe an ndarray in restypes and argtypes specifications. This approach is more flexible than using, for example, `POINTER(c_double)`, since several restrictions can be specified, which are verified upon calling the ctypes function. These include data type, number of dimensions, shape and flags. If a given array does not satisfy the specified restrictions, a `TypeError` is raised.

**Parameters**

**dtype**

[data-type, optional] Array data-type.

**ndim**

[int, optional] Number of array dimensions.

**shape**

[tuple of ints, optional] Array shape.

**flags**

[str or tuple of str] Array flags; may be one or more of:

- C_CONTIGUOUS / C / CONTIGUOUS
- F_CONTIGUOUS / F / FORTRAN
- OWNDATA / O
- WRITEABLE / W
- ALIGNED / A
- WRITEBACKIFCOPY / X
- UPDATEIFCOPY / U

**Returns**

**klass**

[ndpointer type object] A type object, which is an `_ndtpr` instance containing dtype, ndim, shape and flags information.

**Raises**

**TypeError**

If a given array does not satisfy the specified restrictions.

### Examples

```
>>> clib.somefunc.argtypes = [np.ctypeslib.ndpointer(dtype=np.float64,
...                                                  ndim=1,
...                                                  flags='C_CONTIGUOUS')]
...
>>> clib.somefunc(np.array([1, 2, 3], dtype=np.float64))
...
```

## 4.6 Datetime Support Functions

| | |
|---|---|
| `datetime_as_string`(arr[, unit, timezone, …]) | Convert an array of datetimes into an array of strings. |
| `datetime_data`(dtype, /) | Get information about the step size of a date or time type. |

`numpy.`**`datetime_as_string`**(*arr*, *unit=None*, *timezone='naive'*, *casting='same_kind'*)

Convert an array of datetimes into an array of strings.

**Parameters**

**arr**

[array_like of datetime64] The array of UTC timestamps to format.

**unit**

[str] One of None, 'auto', or a *datetime unit*.

**timezone**

[{'naive', 'UTC', 'local'} or tzinfo] Timezone information to use when displaying the datetime. If 'UTC', end with a Z to indicate UTC time. If 'local', convert to the local timezone first, and suffix with a +-#### timezone offset. If a tzinfo object, then do as with 'local', but use the specified timezone.

**casting**

[{'no', 'equiv', 'safe', 'same_kind', 'unsafe'}] Casting to allow when changing between datetime units.

**Returns**

**str_arr**

[ndarray] An array of strings the same shape as *arr*.

### Examples

```
>>> import pytz
>>> d = np.arange('2002-10-27T04:30', 4*60, 60, dtype='M8[m]')
>>> d
array(['2002-10-27T04:30', '2002-10-27T05:30', '2002-10-27T06:30',
       '2002-10-27T07:30'], dtype='datetime64[m]')
```

Setting the timezone to UTC shows the same information, but with a Z suffix

```
>>> np.datetime_as_string(d, timezone='UTC')
array(['2002-10-27T04:30Z', '2002-10-27T05:30Z', '2002-10-27T06:30Z',
       '2002-10-27T07:30Z'], dtype='<U35')
```

Note that we picked datetimes that cross a DST boundary. Passing in a `pytz` timezone object will print the appropriate offset

```
>>> np.datetime_as_string(d, timezone=pytz.timezone('US/Eastern'))
array(['2002-10-27T00:30-0400', '2002-10-27T01:30-0400',
       '2002-10-27T01:30-0500', '2002-10-27T02:30-0500'], dtype='<U39')
```

Passing in a unit will change the precision

```
>>> np.datetime_as_string(d, unit='h')
array(['2002-10-27T04', '2002-10-27T05', '2002-10-27T06', '2002-10-27T07'],
      dtype='<U32')
>>> np.datetime_as_string(d, unit='s')
array(['2002-10-27T04:30:00', '2002-10-27T05:30:00', '2002-10-27T06:30:00',
       '2002-10-27T07:30:00'], dtype='<U38')
```

'casting' can be used to specify whether precision can be changed

```
>>> np.datetime_as_string(d, unit='h', casting='safe')
Traceback (most recent call last):
    ...
TypeError: Cannot create a datetime string as units 'h' from a NumPy
datetime with units 'm' according to the rule 'safe'
```

`numpy.`**`datetime_data`**(*dtype*, */*)

Get information about the step size of a date or time type.

The returned tuple can be passed as the second argument of `numpy.datetime64` and `numpy.timedelta64`.

**Parameters**

**dtype**

[dtype] The dtype object, which must be a `datetime64` or `timedelta64` type.

**Returns**

**unit**

[str] The *datetime unit* on which this dtype is based.

**count**

[int] The number of base units in a step.

### Examples

```
>>> dt_25s = np.dtype('timedelta64[25s]')
>>> np.datetime_data(dt_25s)
('s', 25)
>>> np.array(10, dt_25s).astype('timedelta64[s]')
array(250, dtype='timedelta64[s]')
```

The result can be used to construct a datetime that uses the same units as a timedelta

```
>>> np.datetime64('2010', np.datetime_data(dt_25s))
numpy.datetime64('2010-01-01T00:00:00','25s')
```

## 4.6.1 Business Day Functions

| | |
|---|---|
| *busdaycalendar*([weekmask, holidays]) | A business day calendar object that efficiently stores information defining valid days for the busday family of functions. |
| *is_busday*(dates[, weekmask, holidays, …]) | Calculates which of the given dates are valid days, and which are not. |
| *busday_offset*(dates, offsets[, roll, …]) | First adjusts the date to fall on a valid day according to the `roll` rule, then applies offsets to the given dates counted in valid days. |
| *busday_count*(begindates, enddates[, …]) | Counts the number of valid days between *begindates* and *enddates*, not including the day of *enddates*. |

**class** numpy.**busdaycalendar**(*weekmask='1111100', holidays=None*)

A business day calendar object that efficiently stores information defining valid days for the busday family of functions.

The default valid days are Monday through Friday ("business days"). A busdaycalendar object can be specified with any set of weekly valid days, plus an optional "holiday" dates that always will be invalid.

Once a busdaycalendar object is created, the weekmask and holidays cannot be modified.

New in version 1.7.0.

**Parameters**

**weekmask**

[str or array_like of bool, optional] A seven-element array indicating which of Monday through Sunday are valid days. May be specified as a length-seven list or array, like [1,1,1,1,1,0,0]; a length-seven string, like '1111100'; or a string like "Mon Tue Wed Thu Fri", made up of 3-character abbreviations for weekdays, optionally separated by white space. Valid abbreviations are: Mon Tue Wed Thu Fri Sat Sun

**holidays**

[array_like of datetime64[D], optional] An array of dates to consider as invalid dates, no matter which weekday they fall upon. Holiday dates may be specified in any order, and NaT (not-a-time) dates are ignored. This list is saved in a normalized form that is suited for fast calculations of valid days.

**Returns**

**out**

[busdaycalendar] A business day calendar object containing the specified weekmask and holidays values.

**See also:**

***is_busday***

Returns a boolean array indicating valid days.

***busday_offset***

Applies an offset counted in valid days.

***busday_count***

Counts how many valid days are in a half-open date range.

## Examples

```
>>> # Some important days in July
... bdd = np.busdaycalendar(
...             holidays=['2011-07-01', '2011-07-04', '2011-07-17'])
>>> # Default is Monday to Friday weekdays
... bdd.weekmask
array([ True,  True,  True,  True,  True, False, False])
>>> # Any holidays already on the weekend are removed
... bdd.holidays
array(['2011-07-01', '2011-07-04'], dtype='datetime64[D]')
```

**Attributes**

**Note: once a busdaycalendar object is created, you cannot modify the**

**weekmask or holidays. The attributes return copies of internal data.**

**`weekmask`**

[(copy) seven-element array of bool] A copy of the seven-element boolean mask indicating valid days.

**`holidays`**

[(copy) sorted array of datetime64[D]] A copy of the holiday array indicating additional invalid days.

`numpy.`**`is_busday`**(*dates*, *weekmask='1111100'*, *holidays=None*, *busdaycal=None*, *out=None*)

Calculates which of the given dates are valid days, and which are not.

New in version 1.7.0.

**Parameters**

**dates**

[array_like of datetime64[D]] The array of dates to process.

**weekmask**

[str or array_like of bool, optional] A seven-element array indicating which of Monday through Sunday are valid days. May be specified as a length-seven list or array, like [1,1,1,1,1,0,0]; a length-seven string, like '1111100'; or a string like "Mon Tue Wed Thu Fri", made up of 3-character abbreviations for weekdays, optionally separated by white space. Valid abbreviations are: Mon Tue Wed Thu Fri Sat Sun

**holidays**

[array_like of datetime64[D], optional] An array of dates to consider as invalid dates. They may be specified in any order, and NaT (not-a-time) dates are ignored. This list is saved in a normalized form that is suited for fast calculations of valid days.

**busdaycal**

[busdaycalendar, optional] A `busdaycalendar` object which specifies the valid days. If this parameter is provided, neither weekmask nor holidays may be provided.

**out**

[array of bool, optional] If provided, this array is filled with the result.

**Returns**

**out**

[array of bool] An array with the same shape as `dates`, containing True for each valid day, and False for each invalid day.

**See also:**

**`busdaycalendar`**

An object that specifies a custom set of valid days.

**`busday_offset`**

Applies an offset counted in valid days.

**`busday_count`**

Counts how many valid days are in a half-open date range.

## Examples

```
>>> # The weekdays are Friday, Saturday, and Monday
... np.is_busday(['2011-07-01', '2011-07-02', '2011-07-18'],
...                 holidays=['2011-07-01', '2011-07-04', '2011-07-17'])
array([False, False,  True])
```

`numpy.`**`busday_offset`**(*dates*, *offsets*, *roll='raise'*, *weekmask='1111100'*, *holidays=None*, *busdaycal=None*, *out=None*)

First adjusts the date to fall on a valid day according to the `roll` rule, then applies offsets to the given dates counted in valid days.

New in version 1.7.0.

**Parameters**

**dates**

[array_like of datetime64[D]] The array of dates to process.

**offsets**

[array_like of int] The array of offsets, which is broadcast with `dates`.

**roll**

[{'raise', 'nat', 'forward', 'following', 'backward', 'preceding', 'modifiedfollowing', 'modifiedpreceding'}, optional] How to treat dates that do not fall on a valid day. The default is 'raise'.

- 'raise' means to raise an exception for an invalid day.
- 'nat' means to return a NaT (not-a-time) for an invalid day.
- 'forward' and 'following' mean to take the first valid day later in time.
- 'backward' and 'preceding' mean to take the first valid day earlier in time.
- 'modifiedfollowing' means to take the first valid day later in time unless it is across a Month boundary, in which case to take the first valid day earlier in time.
- 'modifiedpreceding' means to take the first valid day earlier in time unless it is across a Month boundary, in which case to take the first valid day later in time.

**weekmask**

[str or array_like of bool, optional] A seven-element array indicating which of Monday through Sunday are valid days. May be specified as a length-seven list or array, like [1,1,1,1,1,0,0]; a length-seven string, like '1111100'; or a string like "Mon Tue Wed Thu Fri", made up of 3-character abbreviations for weekdays, optionally separated by white space. Valid abbreviations are: Mon Tue Wed Thu Fri Sat Sun

**holidays**

[array_like of datetime64[D], optional] An array of dates to consider as invalid dates. They may be specified in any order, and NaT (not-a-time) dates are ignored. This list is saved in a normalized form that is suited for fast calculations of valid days.

**busdaycal**

[busdaycalendar, optional] A `busdaycalendar` object which specifies the valid days. If this parameter is provided, neither weekmask nor holidays may be provided.

**out**

[array of datetime64[D], optional] If provided, this array is filled with the result.

**Returns**

**out**

[array of datetime64[D]] An array with a shape from broadcasting `dates` and `offsets` together, containing the dates with offsets applied.

**See also:**

**`busdaycalendar`**

An object that specifies a custom set of valid days.

***is_busday***

Returns a boolean array indicating valid days.

***busday_count***

Counts how many valid days are in a half-open date range.

### Examples

```
>>> # First business day in October 2011 (not accounting for holidays)
... np.busday_offset('2011-10', 0, roll='forward')
numpy.datetime64('2011-10-03')
>>> # Last business day in February 2012 (not accounting for holidays)
... np.busday_offset('2012-03', -1, roll='forward')
numpy.datetime64('2012-02-29')
>>> # Third Wednesday in January 2011
... np.busday_offset('2011-01', 2, roll='forward', weekmask='Wed')
numpy.datetime64('2011-01-19')
>>> # 2012 Mother's Day in Canada and the U.S.
... np.busday_offset('2012-05', 1, roll='forward', weekmask='Sun')
numpy.datetime64('2012-05-13')
```

```
>>> # First business day on or after a date
... np.busday_offset('2011-03-20', 0, roll='forward')
numpy.datetime64('2011-03-21')
>>> np.busday_offset('2011-03-22', 0, roll='forward')
numpy.datetime64('2011-03-22')
>>> # First business day after a date
... np.busday_offset('2011-03-20', 1, roll='backward')
numpy.datetime64('2011-03-21')
>>> np.busday_offset('2011-03-22', 1, roll='backward')
numpy.datetime64('2011-03-23')
```

`numpy.`**`busday_count`**(*begindates*, *enddates*, *weekmask='1111100'*, *holidays=[]*, *busdaycal=None*, *out=None*)

Counts the number of valid days between *begindates* and *enddates*, not including the day of *enddates*.

If `enddates` specifies a date value that is earlier than the corresponding `begindates` date value, the count will be negative.

New in version 1.7.0.

**Parameters**

**begindates**

[array_like of datetime64[D]] The array of the first dates for counting.

**enddates**

[array_like of datetime64[D]] The array of the end dates for counting, which are excluded from the count themselves.

**weekmask**

[str or array_like of bool, optional] A seven-element array indicating which of Monday through Sunday are valid days. May be specified as a length-seven list or array, like [1,1,1,1,1,0,0]; a length-seven string, like '1111100'; or a string like "Mon Tue Wed Thu Fri", made up of 3-character abbreviations for weekdays, optionally separated by white space. Valid abbreviations are: Mon Tue Wed Thu Fri Sat Sun

**holidays**

[array_like of datetime64[D], optional] An array of dates to consider as invalid dates. They may be specified in any order, and NaT (not-a-time) dates are ignored. This list is saved in a normalized form that is suited for fast calculations of valid days.

**busdaycal**

[busdaycalendar, optional] A *`busdaycalendar`* object which specifies the valid days. If this parameter is provided, neither weekmask nor holidays may be provided.

**out**

[array of int, optional] If provided, this array is filled with the result.

**Returns**

**out**

[array of int] An array with a shape from broadcasting `begindates` and `enddates` together, containing the number of valid days between the begin and end dates.

**See also:**

***`busdaycalendar`***

An object that specifies a custom set of valid days.

***`is_busday`***

Returns a boolean array indicating valid days.

***`busday_offset`***

Applies an offset counted in valid days.

## Examples

```
>>> # Number of weekdays in January 2011
... np.busday_count('2011-01', '2011-02')
21
>>> # Number of weekdays in 2011
>>> np.busday_count('2011', '2012')
260
>>> # Number of Saturdays in 2011
... np.busday_count('2011', '2012', weekmask='Sat')
53
```

## 4.7 Data type routines

| | |
|---|---|
| `can_cast`(from_, to[, casting]) | Returns True if cast between data types can occur according to the casting rule. |
| `promote_types`(type1, type2) | Returns the data type with the smallest size and smallest scalar kind to which both `type1` and `type2` may be safely cast. |
| `min_scalar_type`(a) | For scalar `a`, returns the data type with the smallest size and smallest scalar kind which can hold its value. |
| `result_type`(*arrays_and_dtypes) | Returns the type that results from applying the NumPy type promotion rules to the arguments. |
| `common_type`(*arrays) | Return a scalar type which is common to the input arrays. |
| `obj2sctype`(rep[, default]) | Return the scalar dtype or NumPy equivalent of Python type of an object. |

`numpy.`**`can_cast`** (*from_*, *to*, *casting='safe'*)

Returns True if cast between data types can occur according to the casting rule. If from is a scalar or array scalar, also returns True if the scalar value can be cast without overflow or truncation to an integer.

**Parameters**

**from_**

[dtype, dtype specifier, scalar, or array] Data type, scalar, or array to cast from.

**to**

[dtype or dtype specifier] Data type to cast to.

**casting**

[{‘no’, ‘equiv’, ‘safe’, ‘same_kind’, ‘unsafe’}, optional] Controls what kind of data casting may occur.

- ‘no’ means the data types should not be cast at all.
- ‘equiv’ means only byte-order changes are allowed.
- ‘safe’ means only casts which can preserve values are allowed.
- ‘same_kind’ means only safe casts or casts within a kind, like float64 to float32, are allowed.
- ‘unsafe’ means any data conversions may be done.

**Returns**

**out**

[bool] True if cast can occur according to the casting rule.

**See also:**

`dtype`, `result_type`

## Notes

Changed in version 1.17.0: Casting between a simple data type and a structured one is possible only for “unsafe” casting. Casting to multiple fields is allowed, but casting from multiple fields is not.

Changed in version 1.9.0: Casting from numeric to string types in ‘safe’ casting mode requires that the string dtype length is long enough to store the maximum integer/float value converted.

## Examples

Basic examples

```
>>> np.can_cast(np.int32, np.int64)
True
>>> np.can_cast(np.float64, complex)
True
>>> np.can_cast(complex, float)
False
```

```
>>> np.can_cast('i8', 'f8')
True
>>> np.can_cast('i8', 'f4')
False
>>> np.can_cast('i4', 'S4')
False
```

Casting scalars

```
>>> np.can_cast(100, 'i1')
True
>>> np.can_cast(150, 'i1')
False
>>> np.can_cast(150, 'u1')
True
```

```
>>> np.can_cast(3.5e100, np.float32)
False
>>> np.can_cast(1000.0, np.float32)
True
```

Array scalar checks the value, array does not

```
>>> np.can_cast(np.array(1000.0), np.float32)
True
>>> np.can_cast(np.array([1000.0]), np.float32)
False
```

Using the casting rules

```
>>> np.can_cast('i8', 'i8', 'no')
True
>>> np.can_cast('<i8', '>i8', 'no')
False
```

```
>>> np.can_cast('<i8', '>i8', 'equiv')
True
>>> np.can_cast('<i4', '>i8', 'equiv')
False
```

```
>>> np.can_cast('<i4', '>i8', 'safe')
True
>>> np.can_cast('<i8', '>i4', 'safe')
False
```

```
>>> np.can_cast('<i8', '>i4', 'same_kind')
True
>>> np.can_cast('<i8', '>u4', 'same_kind')
False
```

```
>>> np.can_cast('<i8', '>u4', 'unsafe')
True
```

`numpy.`**`promote_types`**(*type1*, *type2*)

Returns the data type with the smallest size and smallest scalar kind to which both `type1` and `type2` may be safely cast. The returned data type is always in native byte order.

This function is symmetric, but rarely associative.

**Parameters**

**type1**

[dtype or dtype specifier] First data type.

**type2**

[dtype or dtype specifier] Second data type.

**Returns**

**out**

[dtype] The promoted data type.

**See also:**

*`result_type`*, *`dtype`*, *`can_cast`*

## Notes

New in version 1.6.0.

Starting in NumPy 1.9, promote_types function now returns a valid string length when given an integer or float dtype as one argument and a string dtype as another argument. Previously it always returned the input string dtype, even if it wasn't long enough to store the max integer/float value converted to a string.

## Examples

```
>>> np.promote_types('f4', 'f8')
dtype('float64')
```

```
>>> np.promote_types('i8', 'f4')
dtype('float64')
```

```
>>> np.promote_types('>i8', '<c8')
dtype('complex128')
```

```
>>> np.promote_types('i4', 'S8')
dtype('S11')
```

An example of a non-associative case:

```
>>> p = np.promote_types
>>> p('S', p('i1', 'u1'))
dtype('S6')
>>> p(p('S', 'i1'), 'u1')
dtype('S4')
```

`numpy.`**`min_scalar_type`**`(`*a*`)`

For scalar `a`, returns the data type with the smallest size and smallest scalar kind which can hold its value. For non-scalar array `a`, returns the vector's dtype unmodified.

Floating point values are not demoted to integers, and complex values are not demoted to floats.

**Parameters**

**a**

[scalar or array_like] The value whose minimal data type is to be found.

**Returns**

**out**

[dtype] The minimal data type.

**See also:**

*`result_type`*, *`promote_types`*, *`dtype`*, *`can_cast`*

## Notes

New in version 1.6.0.

## Examples

```
>>> np.min_scalar_type(10)
dtype('uint8')
```

```
>>> np.min_scalar_type(-260)
dtype('int16')
```

```
>>> np.min_scalar_type(3.1)
dtype('float16')
```

```
>>> np.min_scalar_type(1e50)
dtype('float64')
```

```
>>> np.min_scalar_type(np.arange(4,dtype='f8'))
dtype('float64')
```

`numpy.`**`result_type`**`(`**arrays_and_dtypes*`)`

Returns the type that results from applying the NumPy type promotion rules to the arguments.

Type promotion in NumPy works similarly to the rules in languages like C++, with some slight differences. When both scalars and arrays are used, the array's type takes precedence and the actual value of the scalar is taken into account.

For example, calculating 3*a, where a is an array of 32-bit floats, intuitively should result in a 32-bit float output. If the 3 is a 32-bit integer, the NumPy rules indicate it can't convert losslessly into a 32-bit float, so a 64-bit float should be the result type. By examining the value of the constant, '3', we see that it fits in an 8-bit integer, which can be cast losslessly into the 32-bit float.

**Parameters**

**arrays_and_dtypes**

[list of arrays and dtypes] The operands of some operation whose result type is needed.

**Returns**

**out**

[dtype] The result type.

**See also:**

`dtype`, `promote_types`, `min_scalar_type`, `can_cast`

## Notes

New in version 1.6.0.

The specific algorithm used is as follows.

Categories are determined by first checking which of boolean, integer (int/uint), or floating point (float/complex) the maximum kind of all the arrays and the scalars are.

If there are only scalars or the maximum category of the scalars is higher than the maximum category of the arrays, the data types are combined with `promote_types` to produce the return value.

Otherwise, *min_scalar_type* is called on each array, and the resulting data types are all combined with *promote_types* to produce the return value.

The set of int values is not a subset of the uint values for types with the same number of bits, something not reflected in *min_scalar_type*, but handled as a special case in *result_type*.

### Examples

```
>>> np.result_type(3, np.arange(7, dtype='i1'))
dtype('int8')
```

```
>>> np.result_type('i4', 'c8')
dtype('complex128')
```

```
>>> np.result_type(3.0, -2)
dtype('float64')
```

`numpy.`**`common_type`**`(`**arrays*`)`

Return a scalar type which is common to the input arrays.

The return type will always be an inexact (i.e. floating point) scalar type, even if all the arrays are integer arrays. If one of the inputs is an integer array, the minimum precision type that is returned is a 64-bit floating point dtype.

All input arrays except int64 and uint64 can be safely cast to the returned dtype without loss of information.

**Parameters**

**array1, array2, …**

[ndarrays] Input arrays.

**Returns**

**out**

[data type code] Data type code.

**See also:**

*dtype*, *mintypecode*

### Examples

```
>>> np.common_type(np.arange(2, dtype=np.float32))
<class 'numpy.float32'>
>>> np.common_type(np.arange(2, dtype=np.float32), np.arange(2))
<class 'numpy.float64'>
>>> np.common_type(np.arange(4), np.array([45, 6.j]), np.array([45.0]))
<class 'numpy.complex128'>
```

`numpy.`**`obj2sctype`**`(`*rep*, *default=None*`)`

Return the scalar dtype or NumPy equivalent of Python type of an object.

**Parameters**

**rep**

[any] The object of which the type is returned.

**default**

[any, optional] If given, this is returned for objects whose types can not be determined. If not given, None is returned for those objects.

**Returns**

**dtype**

[dtype or Python type] The data type of *rep*.

**See also:**

*sctype2char*, *issctype*, *issubsctype*, *issubdtype*, *maximum_sctype*

### Examples

```
>>> np.obj2sctype(np.int32)
<class 'numpy.int32'>
>>> np.obj2sctype(np.array([1., 2.]))
<class 'numpy.float64'>
>>> np.obj2sctype(np.array([1.j]))
<class 'numpy.complex128'>
```

```
>>> np.obj2sctype(dict)
<class 'numpy.object_'>
>>> np.obj2sctype('string')
```

```
>>> np.obj2sctype(1, default=list)
<class 'list'>
```

## 4.7.1 Creating data types

| | |
|---|---|
| *dtype*(obj[, align, copy]) | Create a data type object. |
| *format_parser*(formats, names, titles[, …]) | Class to convert formats, names, titles description to a dtype. |

**class** `numpy.`**`format_parser`**(*formats*, *names*, *titles*, *aligned=False*, *byteorder=None*)

Class to convert formats, names, titles description to a dtype.

After constructing the format_parser object, the dtype attribute is the converted data-type: `dtype = format_parser(formats, names, titles).dtype`

**Parameters**

**formats**

[str or list of str] The format description, either specified as a string with comma-separated format descriptions in the form `'f8, i4, a5'`, or a list of format description strings in the form `['f8', 'i4', 'a5']`.

**names**

[str or list/tuple of str] The field names, either specified as a comma-separated string in the form `'col1, col2, col3'`, or as a list or tuple of strings in the form `['col1', 'col2', 'col3']`. An empty list can be used, in that case default field names ('f0', 'f1', …) are used.

**titles**

[sequence] Sequence of title strings. An empty list can be used to leave titles out.

**aligned**

[bool, optional] If True, align the fields by padding as the C-compiler would. Default is False.

**byteorder**

[str, optional] If specified, all the fields will be changed to the provided byte-order. Otherwise, the default byte-order is used. For all available string specifiers, see `dtype.newbyteorder`.

**See also:**

`dtype`, `typename`, `sctype2char`

### Examples

```
>>> np.format_parser(['<f8', '<i4', '<a5'], ['col1', 'col2', 'col3'],
...                  ['T1', 'T2', 'T3']).dtype
dtype([(('T1', 'col1'), '<f8'), (('T2', 'col2'), '<i4'), (('T3', 'col3'), 'S5')])
```

*names* and/or *titles* can be empty lists. If *titles* is an empty list, titles will simply not appear. If *names* is empty, default field names will be used.

```
>>> np.format_parser(['f8', 'i4', 'a5'], ['col1', 'col2', 'col3'],
...                  []).dtype
dtype([('col1', '<f8'), ('col2', '<i4'), ('col3', '<S5')])
>>> np.format_parser(['<f8', '<i4', '<a5'], [], []).dtype
dtype([('f0', '<f8'), ('f1', '<i4'), ('f2', 'S5')])
```

**Attributes**

**dtype**

[dtype] The converted data-type.

## 4.7.2 Data type information

| | |
|---|---|
| `finfo`(dtype) | Machine limits for floating point types. |
| `iinfo`(type) | Machine limits for integer types. |
| `MachAr`([float_conv, int_conv, …]) | Diagnosing machine parameters. |

**class** `numpy.`**`finfo`**(*dtype*)

Machine limits for floating point types.

**Parameters**

**dtype**

[float, dtype, or instance] Kind of floating point data-type about which to get information.

**See also:**

*`MachAr`*

The implementation of the tests that produce this information.

*`iinfo`*

The equivalent for integer data types.

*`spacing`*

The distance between a value and the nearest adjacent number

*`nextafter`*

The next floating point value after x1 towards x2

## Notes

For developers of NumPy: do not instantiate this at the module level. The initial calculation of these parameters is expensive and negatively impacts import times. These objects are cached, so calling `finfo()` repeatedly inside your functions is not a problem.

**Attributes**

**bits**

[int] The number of bits occupied by the type.

**eps**

[float] The difference between 1.0 and the next smallest representable float larger than 1.0. For example, for 64-bit binary floats in the IEEE-754 standard, `eps = 2**-52`, approximately 2.22e-16.

**epsneg**

[float] The difference between 1.0 and the next smallest representable float less than 1.0. For example, for 64-bit binary floats in the IEEE-754 standard, `epsneg = 2**-53`, approximately 1.11e-16.

**iexp**

[int] The number of bits in the exponent portion of the floating point representation.

**machar**

[MachAr] The object which calculated these parameters and holds more detailed information.

**machep**

[int] The exponent that yields *eps*.

**max**

[floating point number of the appropriate type] The largest representable number.

**maxexp**

[int] The smallest positive power of the base (2) that causes overflow.

**min**

[floating point number of the appropriate type] The smallest representable number, typically `-max`.

**minexp**

[int] The most negative power of the base (2) consistent with there being no leading 0's in the mantissa.

**negep**

[int] The exponent that yields *epsneg*.

**nexp**

[int] The number of bits in the exponent including its sign and bias.

**nmant**

[int] The number of bits in the mantissa.

**precision**

[int] The approximate number of decimal digits to which this kind of float is precise.

**resolution**

[floating point number of the appropriate type] The approximate decimal resolution of this type, i.e., `10**-precision`.

**tiny**

[float] The smallest positive usable number. Type of *tiny* is an appropriate floating point type.

`class numpy.iinfo(type)`

Machine limits for integer types.

**Parameters**

**int_type**

[integer type, dtype, or instance] The kind of integer data type to get information about.

**See also:**

***finfo***

The equivalent for floating point data types.

### Examples

With types:

```
>>> ii16 = np.iinfo(np.int16)
>>> ii16.min
-32768
>>> ii16.max
32767
>>> ii32 = np.iinfo(np.int32)
>>> ii32.min
-2147483648
```

(continues on next page)

(continued from previous page)

```
>>> ii32.max
2147483647
```

With instances:

```
>>> ii32 = np.iinfo(np.int32(10))
>>> ii32.min
-2147483648
>>> ii32.max
2147483647
```

**Attributes**

**bits**

[int] The number of bits occupied by the type.

**`min`**

[int] Minimum value of given dtype.

**`max`**

[int] Maximum value of given dtype.

**`class`** `numpy.`**`MachAr`**(*float_conv=<class 'float'>, int_conv=<class 'int'>, float_to_float=<class 'float'>, float_to_str=<function MachAr.<lambda», title='Python floating point number'*)

Diagnosing machine parameters.

**Parameters**

**float_conv**

[function, optional] Function that converts an integer or integer array to a float or float array. Default is `float`.

**int_conv**

[function, optional] Function that converts a float or float array to an integer or integer array. Default is `int`.

**float_to_float**

[function, optional] Function that converts a float array to float. Default is `float`. Note that this does not seem to do anything useful in the current implementation.

**float_to_str**

[function, optional] Function that converts a single float to a string. Default is `lambda v:'%24.16e' %v`.

**title**

[str, optional] Title that is printed in the string representation of *`MachAr`*.

**See also:**

***`finfo`***

Machine limits for floating point types.

*iinfo*

Machine limits for integer types.

## References

[Re860718f5533-1]

**Attributes**

**ibeta**

[int] Radix in which numbers are represented.

**it**

[int] Number of base-*ibeta* digits in the floating point mantissa M.

**machep**

[int] Exponent of the smallest (most negative) power of *ibeta* that, added to 1.0, gives something different from 1.0

**eps**

[float] Floating-point number `beta**machep` (floating point precision)

**negep**

[int] Exponent of the smallest power of *ibeta* that, subtracted from 1.0, gives something different from 1.0.

**epsneg**

[float] Floating-point number `beta**negep`.

**iexp**

[int] Number of bits in the exponent (including its sign and bias).

**minexp**

[int] Smallest (most negative) power of *ibeta* consistent with there being no leading zeros in the mantissa.

**xmin**

[float] Floating point number `beta**minexp` (the smallest [in magnitude] usable floating value).

**maxexp**

[int] Smallest (positive) power of *ibeta* that causes overflow.

**xmax**

[float] `(1-epsneg) * beta**maxexp` (the largest [in magnitude] usable floating value).

**irnd**

[int] In `range(6)`, information on what kind of rounding is done in addition, and on how underflow is handled.

**ngrd**

[int] Number of 'guard digits' used when truncating the product of two mantissas to fit the representation.

**epsilon**

[float] Same as *eps*.

**tiny**

[float] Same as *xmin*.

**huge**

[float] Same as *xmax*.

**precision**

[float] – `int(-log10(eps))`

**resolution**

[float] – `10**(-precision)`

### 4.7.3 Data type testing

| | |
|---|---|
| `issctype`(rep) | Determines whether the given object represents a scalar data-type. |
| `issubdtype`(arg1, arg2) | Returns True if first argument is a typecode lower/equal in type hierarchy. |
| `issubsctype`(arg1, arg2) | Determine if the first argument is a subclass of the second argument. |
| `issubclass_`(arg1, arg2) | Determine if a class is a subclass of a second class. |
| `find_common_type`(array_types, scalar_types) | Determine common type following standard coercion rules. |

`numpy.`**`issctype`**(*rep*)

Determines whether the given object represents a scalar data-type.

**Parameters**

**rep**

[any] If *rep* is an instance of a scalar dtype, True is returned. If not, False is returned.

**Returns**

**out**

[bool] Boolean result of check whether *rep* is a scalar dtype.

**See also:**

`issubsctype`, `issubdtype`, `obj2sctype`, `sctype2char`

## Examples

```
>>> np.issctype(np.int32)
True
>>> np.issctype(list)
False
>>> np.issctype(1.1)
False
```

Strings are also a scalar type:

```
>>> np.issctype(np.dtype('str'))
True
```

`numpy.`**`issubdtype`**(*arg1*, *arg2*)

Returns True if first argument is a typecode lower/equal in type hierarchy.

**Parameters**

**arg1, arg2**

[dtype_like] dtype or string representing a typecode.

**Returns**

**out**

[bool]

**See also:**

*`issubsctype`*, *`issubclass_`*

**`numpy.core.numerictypes`**

Overview of numpy type hierarchy.

## Examples

```
>>> np.issubdtype('S1', np.string_)
True
>>> np.issubdtype(np.float64, np.float32)
False
```

`numpy.`**`issubsctype`**(*arg1*, *arg2*)

Determine if the first argument is a subclass of the second argument.

**Parameters**

**arg1, arg2**

[dtype or dtype specifier] Data-types.

**Returns**

**out**

[bool] The result.

**See also:**

*issctype*, *issubdtype*, *obj2sctype*

## Examples

```
>>> np.issubsctype('S8', str)
False
>>> np.issubsctype(np.array([1]), int)
True
>>> np.issubsctype(np.array([1]), float)
False
```

`numpy.`**`issubclass_`**`(`*arg1*, *arg2*`)`

Determine if a class is a subclass of a second class.

*issubclass_* is equivalent to the Python built-in `issubclass`, except that it returns False instead of raising a TypeError if one of the arguments is not a class.

**Parameters**

**arg1**

[class] Input class. True is returned if *arg1* is a subclass of *arg2*.

**arg2**

[class or tuple of classes.] Input class. If a tuple of classes, True is returned if *arg1* is a subclass of any of the tuple elements.

**Returns**

**out**

[bool] Whether *arg1* is a subclass of *arg2* or not.

**See also:**

*issubsctype*, *issubdtype*, *issctype*

## Examples

```
>>> np.issubclass_(np.int32, int)
False
>>> np.issubclass_(np.int32, float)
False
>>> np.issubclass_(np.float64, float)
True
```

`numpy.`**`find_common_type`**`(`*array_types*, *scalar_types*`)`

Determine common type following standard coercion rules.

**Parameters**

**array_types**

[sequence] A list of dtypes or dtype convertible objects representing arrays.

**scalar_types**

[sequence] A list of dtypes or dtype convertible objects representing scalars.

**Returns**

**datatype**

[dtype] The common data type, which is the maximum of *array_types* ignoring *scalar_types*, unless the maximum of *scalar_types* is of a different kind (`dtype.kind`). If the kind is not understood, then None is returned.

**See also:**

`dtype`, `common_type`, `can_cast`, `mintypecode`

### Examples

```
>>> np.find_common_type([], [np.int64, np.float32, complex])
dtype('complex128')
>>> np.find_common_type([np.int64, np.float32], [])
dtype('float64')
```

The standard casting rules ensure that a scalar cannot up-cast an array unless the scalar is of a fundamentally different kind of data (i.e. under a different hierarchy in the data type hierarchy) then the array:

```
>>> np.find_common_type([np.float32], [np.int64, np.float64])
dtype('float32')
```

Complex is of a different type, so it up-casts the float in the *array_types* argument:

```
>>> np.find_common_type([np.float32], [complex])
dtype('complex128')
```

Type specifier strings are convertible to dtypes and can therefore be used instead of dtypes:

```
>>> np.find_common_type(['f4', 'f4', 'i4'], ['c8'])
dtype('complex128')
```

## 4.7.4 Miscellaneous

| | |
|---|---|
| `typename`(char) | Return a description for the given data type code. |
| `sctype2char`(sctype) | Return the string representation of a scalar dtype. |
| `mintypecode`(typechars[, typeset, default]) | Return the character for the minimum-size type to which given types can be safely cast. |
| `maximum_sctype`(t) | Return the scalar type of highest precision of the same kind as the input. |

`numpy.`**`typename`**(*char*)

Return a description for the given data type code.

**Parameters**

**char**

[str] Data type code.

**Returns**

**out**

[str] Description of the input data type code.

**See also:**

*dtype*, `typecodes`

### Examples

```
>>> typechars = ['S1', '?', 'B', 'D', 'G', 'F', 'I', 'H', 'L', 'O', 'Q',
...              'S', 'U', 'V', 'b', 'd', 'g', 'f', 'i', 'h', 'l', 'q']
>>> for typechar in typechars:
...     print(typechar, ' : ', np.typename(typechar))
...
S1  :  character
?  :  bool
B  :  unsigned char
D  :  complex double precision
G  :  complex long double precision
F  :  complex single precision
I  :  unsigned integer
H  :  unsigned short
L  :  unsigned long integer
O  :  object
Q  :  unsigned long long integer
S  :  string
U  :  unicode
V  :  void
b  :  signed char
d  :  double precision
g  :  long precision
f  :  single precision
i  :  integer
h  :  short
l  :  long integer
q  :  long long integer
```

`numpy.`**`sctype2char`**(*sctype*)

Return the string representation of a scalar dtype.

**Parameters**

**sctype**

[scalar dtype or object] If a scalar dtype, the corresponding string character is returned. If an object, *sctype2char* tries to infer its scalar type and then return the corresponding string character.

**Returns**

**typechar**

[str] The string character corresponding to the scalar type.

**Raises**

**ValueError**

If *sctype* is an object for which the type can not be inferred.

**See also:**

`obj2sctype`, `issctype`, `issubsctype`, `mintypecode`

## Examples

```
>>> for sctype in [np.int32, np.double, np.complex_, np.string_, np.ndarray]:
...     print(np.sctype2char(sctype))
l # may vary
d
D
S
O
```

```
>>> x = np.array([1., 2-1.j])
>>> np.sctype2char(x)
'D'
>>> np.sctype2char(list)
'O'
```

`numpy.`**`mintypecode`**(*typechars*, *typeset='GDFgdf'*, *default='d'*)

Return the character for the minimum-size type to which given types can be safely cast.

The returned type character must represent the smallest size dtype such that an array of the returned type can handle the data from an array of all types in *typechars* (or if *typechars* is an array, then its dtype.char).

**Parameters**

**typechars**

[list of str or array_like] If a list of strings, each string should represent a dtype. If array_like, the character representation of the array dtype is used.

**typeset**

[str or list of str, optional] The set of characters that the returned character is chosen from. The default set is 'GDFgdf'.

**default**

[str, optional] The default character, this is returned if none of the characters in *typechars* matches a character in *typeset*.

**Returns**

**typechar**

[str] The character representing the minimum-size type that was found.

**See also:**

`dtype`, `sctype2char`, `maximum_sctype`

## Examples

```
>>> np.mintypecode(['d', 'f', 'S'])
'd'
>>> x = np.array([1.1, 2-3.j])
>>> np.mintypecode(x)
'D'
```

```
>>> np.mintypecode('abceh', default='G')
'G'
```

`numpy.`**`maximum_sctype`**`(`*t*`)`

Return the scalar type of highest precision of the same kind as the input.

**Parameters**

**t**

[dtype or dtype specifier] The input data type. This can be a *`dtype`* object or an object that is convertible to a *`dtype`*.

**Returns**

**out**

[dtype] The highest precision data type of the same kind (*`dtype.kind`*) as *t*.

**See also:**

*`obj2sctype`*, *`mintypecode`*, *`sctype2char`*, *`dtype`*

## Examples

```
>>> np.maximum_sctype(int)
<class 'numpy.int64'>
>>> np.maximum_sctype(np.uint8)
<class 'numpy.uint64'>
>>> np.maximum_sctype(complex)
<class 'numpy.complex256'> # may vary
```

```
>>> np.maximum_sctype(str)
<class 'numpy.str_'>
```

```
>>> np.maximum_sctype('i2')
<class 'numpy.int64'>
>>> np.maximum_sctype('f4')
<class 'numpy.float128'> # may vary
```

## 4.8 Optionally Scipy-accelerated routines (`numpy.dual`)

Aliases for functions which may be accelerated by Scipy.

Scipy can be built to use accelerated or otherwise improved libraries for FFTs, linear algebra, and special functions. This module allows developers to transparently support these accelerated functions when scipy is available but still support users who have only installed NumPy.

### 4.8.1 Linear algebra

| | |
|---|---|
| `cholesky`(a) | Cholesky decomposition. |
| `det`(a) | Compute the determinant of an array. |
| `eig`(a) | Compute the eigenvalues and right eigenvectors of a square array. |
| `eigh`(a[, UPLO]) | Return the eigenvalues and eigenvectors of a complex Hermitian (conjugate symmetric) or a real symmetric matrix. |
| `eigvals`(a) | Compute the eigenvalues of a general matrix. |
| `eigvalsh`(a[, UPLO]) | Compute the eigenvalues of a complex Hermitian or real symmetric matrix. |
| `inv`(a) | Compute the (multiplicative) inverse of a matrix. |
| `lstsq`(a, b[, rcond]) | Return the least-squares solution to a linear matrix equation. |
| `norm`(x[, ord, axis, keepdims]) | Matrix or vector norm. |
| `pinv`(a[, rcond, hermitian]) | Compute the (Moore-Penrose) pseudo-inverse of a matrix. |
| `solve`(a, b) | Solve a linear matrix equation, or system of linear scalar equations. |
| `svd`(a[, full_matrices, compute_uv, hermitian]) | Singular Value Decomposition. |

### 4.8.2 FFT

| | |
|---|---|
| `fft`(a[, n, axis, norm]) | Compute the one-dimensional discrete Fourier Transform. |
| `fft2`(a[, s, axes, norm]) | Compute the 2-dimensional discrete Fourier Transform |
| `fftn`(a[, s, axes, norm]) | Compute the N-dimensional discrete Fourier Transform. |
| `ifft`(a[, n, axis, norm]) | Compute the one-dimensional inverse discrete Fourier Transform. |
| `ifft2`(a[, s, axes, norm]) | Compute the 2-dimensional inverse discrete Fourier Transform. |
| `ifftn`(a[, s, axes, norm]) | Compute the N-dimensional inverse discrete Fourier Transform. |

### 4.8.3 Other

| | |
|---|---|
| *`i0`*(x) | Modified Bessel function of the first kind, order 0. |

## 4.9 Mathematical functions with automatic domain (`numpy.emath`)

**Note:** `numpy.emath` is a preferred alias for *`numpy.lib.scimath`*, available after *`numpy`* is imported.

Wrapper functions to more user-friendly calling of certain math functions whose output data-type is different than the input data-type in certain domains of the input.

For example, for functions like `log` with branch cuts, the versions in this module provide the mathematically valid answers in the complex plane:

```
>>> import math
>>> from numpy.lib import scimath
>>> scimath.log(-math.exp(1)) == (1+1j*math.pi)
True
```

Similarly, `sqrt`, other base logarithms, `power` and trig functions are correctly handled. See their respective docstrings for specific examples.

## 4.10 Floating point error handling

### 4.10.1 Setting and getting error handling

| | |
|---|---|
| *`seterr`*([all, divide, over, under, invalid]) | Set how floating-point errors are handled. |
| *`geterr`*() | Get the current way of handling floating-point errors. |
| *`seterrcall`*(func) | Set the floating-point error callback function or log object. |
| *`geterrcall`*() | Return the current callback function used on floating-point errors. |
| *`errstate`*(**kwargs) | Context manager for floating-point error handling. |

`numpy.`**`geterr`**`()`

Get the current way of handling floating-point errors.

**Returns**

**res**

[dict] A dictionary with keys "divide", "over", "under", and "invalid", whose values are from the strings "ignore", "print", "log", "warn", "raise", and "call". The keys represent possible floating-point exceptions, and the values define how these exceptions are handled.

**See also:**

*`geterrcall`*, *`seterr`*, *`seterrcall`*

## Notes

For complete documentation of the types of floating-point exceptions and treatment options, see *seterr*.

## Examples

```
>>> from collections import OrderedDict
>>> sorted(np.geterr().items())
[('divide', 'warn'), ('invalid', 'warn'), ('over', 'warn'), ('under', 'ignore')]
>>> np.arange(3.) / np.arange(3.)
array([nan,  1.,  1.])
```

```
>>> oldsettings = np.seterr(all='warn', over='raise')
>>> OrderedDict(sorted(np.geterr().items()))
OrderedDict([('divide', 'warn'), ('invalid', 'warn'), ('over', 'raise'), ('under',
↪ 'warn')])
>>> np.arange(3.) / np.arange(3.)
array([nan,  1.,  1.])
```

`numpy.`**`geterrcall`**`()`

Return the current callback function used on floating-point errors.

When the error handling for a floating-point error (one of "divide", "over", "under", or "invalid") is set to 'call' or 'log', the function that is called or the log instance that is written to is returned by *geterrcall*. This function or log instance has been set with *seterrcall*.

**Returns**

**errobj**

[callable, log instance or None] The current error handler. If no handler was set through *seterrcall*, `None` is returned.

**See also:**

*seterrcall*, *seterr*, *geterr*

## Notes

For complete documentation of the types of floating-point exceptions and treatment options, see *seterr*.

## Examples

```
>>> np.geterrcall()  # we did not yet set a handler, returns None
```

```
>>> oldsettings = np.seterr(all='call')
>>> def err_handler(type, flag):
...     print("Floating point error (%s), with flag %s" % (type, flag))
>>> oldhandler = np.seterrcall(err_handler)
>>> np.array([1, 2, 3]) / 0.0
Floating point error (divide by zero), with flag 1
array([inf, inf, inf])
```

```
>>> cur_handler = np.geterrcall()
>>> cur_handler is err_handler
True
```

**class** `numpy.errstate`(***kwargs*)

Context manager for floating-point error handling.

Using an instance of *errstate* as a context manager allows statements in that context to execute with a known error handling behavior. Upon entering the context the error handling is set with *seterr* and *seterrcall*, and upon exiting it is reset to what it was before.

Changed in version 1.17.0: *errstate* is also usable as a function decorator, saving a level of indentation if an entire function is wrapped. See `contextlib.ContextDecorator` for more information.

**Parameters**

**kwargs**

[{divide, over, under, invalid}] Keyword arguments. The valid keywords are the possible floating-point exceptions. Each keyword should have a string value that defines the treatment for the particular error. Possible values are {'ignore', 'warn', 'raise', 'call', 'print', 'log'}.

**See also:**

*seterr*, *geterr*, *seterrcall*, *geterrcall*

### Notes

For complete documentation of the types of floating-point exceptions and treatment options, see *seterr*.

### Examples

```
>>> from collections import OrderedDict
>>> olderr = np.seterr(all='ignore')  # Set error handling to known state.
```

```
>>> np.arange(3) / 0.
array([nan, inf, inf])
>>> with np.errstate(divide='warn'):
...     np.arange(3) / 0.
array([nan, inf, inf])
```

```
>>> np.sqrt(-1)
nan
>>> with np.errstate(invalid='raise'):
...     np.sqrt(-1)
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
FloatingPointError: invalid value encountered in sqrt
```

Outside the context the error handling behavior has not changed:

```
>>> OrderedDict(sorted(np.geterr().items()))
OrderedDict([('divide', 'ignore'), ('invalid', 'ignore'), ('over', 'ignore'), (
↪'under', 'ignore')])
```

### Methods

| | |
|---|---|
| `__call__`(self, func) | Call self as a function. |

method

`errstate.`**`__call__`**(*self*, *func*)
Call self as a function.

## 4.10.2 Internal functions

| | |
|---|---|
| `seterrobj`(errobj) | Set the object that defines floating-point error handling. |
| `geterrobj`() | Return the current object that defines floating-point error handling. |

`numpy.`**`seterrobj`**(*errobj*)
Set the object that defines floating-point error handling.

The error object contains all information that defines the error handling behavior in NumPy. `seterrobj` is used internally by the other functions that set error handling behavior (`seterr`, `seterrcall`).

**Parameters**

**errobj**

[list] The error object, a list containing three elements: [internal numpy buffer size, error mask, error callback function].

The error mask is a single integer that holds the treatment information on all four floating point errors. The information for each error type is contained in three bits of the integer. If we print it in base 8, we can see what treatment is set for “invalid”, “under”, “over”, and “divide” (in that order). The printed string can be interpreted with

- 0 : ‘ignore’
- 1 : ‘warn’
- 2 : ‘raise’
- 3 : ‘call’
- 4 : ‘print’
- 5 : ‘log’

**See also:**

`geterrobj`, `seterr`, `geterr`, `seterrcall`, `geterrcall`, `getbufsize`, `setbufsize`

## Notes

For complete documentation of the types of floating-point exceptions and treatment options, see *seterr*.

## Examples

```
>>> old_errobj = np.geterrobj()  # first get the defaults
>>> old_errobj
[8192, 521, None]
```

```
>>> def err_handler(type, flag):
...     print("Floating point error (%s), with flag %s" % (type, flag))
...
>>> new_errobj = [20000, 12, err_handler]
>>> np.seterrobj(new_errobj)
>>> np.base_repr(12, 8)  # int for divide=4 ('print') and over=1 ('warn')
'14'
>>> np.geterr()
{'over': 'warn', 'divide': 'print', 'invalid': 'ignore', 'under': 'ignore'}
>>> np.geterrcall() is err_handler
True
```

`numpy.`**`geterrobj`**`()`

Return the current object that defines floating-point error handling.

The error object contains all information that defines the error handling behavior in NumPy. *geterrobj* is used internally by the other functions that get and set error handling behavior (*geterr*, *seterr*, *geterrcall*, *seterrcall*).

**Returns**

**errobj**

[list] The error object, a list containing three elements: [internal numpy buffer size, error mask, error callback function].

The error mask is a single integer that holds the treatment information on all four floating point errors. The information for each error type is contained in three bits of the integer. If we print it in base 8, we can see what treatment is set for “invalid”, “under”, “over”, and “divide” (in that order). The printed string can be interpreted with

- 0 : ‘ignore’
- 1 : ‘warn’
- 2 : ‘raise’
- 3 : ‘call’
- 4 : ‘print’
- 5 : ‘log’

**See also:**

*seterrobj*, *seterr*, *geterr*, *seterrcall*, *geterrcall*, *getbufsize*, *setbufsize*

### Notes

For complete documentation of the types of floating-point exceptions and treatment options, see `seterr`.

### Examples

```
>>> np.geterrobj()  # first get the defaults
[8192, 521, None]
```

```
>>> def err_handler(type, flag):
...     print("Floating point error (%s), with flag %s" % (type, flag))
...
>>> old_bufsize = np.setbufsize(20000)
>>> old_err = np.seterr(divide='raise')
>>> old_handler = np.seterrcall(err_handler)
>>> np.geterrobj()
[8192, 521, <function err_handler at 0x91dcaac>]
```

```
>>> old_err = np.seterr(all='ignore')
>>> np.base_repr(np.geterrobj()[1], 8)
'0'
>>> old_err = np.seterr(divide='warn', over='log', under='call',
...                     invalid='print')
>>> np.base_repr(np.geterrobj()[1], 8)
'4351'
```

## 4.11 Discrete Fourier Transform (`numpy.fft`)

### 4.11.1 Standard FFTs

| | |
|---|---|
| *fft*(a[, n, axis, norm]) | Compute the one-dimensional discrete Fourier Transform. |
| *ifft*(a[, n, axis, norm]) | Compute the one-dimensional inverse discrete Fourier Transform. |
| *fft2*(a[, s, axes, norm]) | Compute the 2-dimensional discrete Fourier Transform |
| *ifft2*(a[, s, axes, norm]) | Compute the 2-dimensional inverse discrete Fourier Transform. |
| *fftn*(a[, s, axes, norm]) | Compute the N-dimensional discrete Fourier Transform. |
| *ifftn*(a[, s, axes, norm]) | Compute the N-dimensional inverse discrete Fourier Transform. |

`numpy.fft.`**`fft`**(*a*, *n=None*, *axis=-1*, *norm=None*)

Compute the one-dimensional discrete Fourier Transform.

This function computes the one-dimensional *n*-point discrete Fourier Transform (DFT) with the efficient Fast Fourier Transform (FFT) algorithm [CT].

**Parameters**

**a**

[array_like] Input array, can be complex.

**n**

[int, optional] Length of the transformed axis of the output. If *n* is smaller than the length of the input, the input is cropped. If it is larger, the input is padded with zeros. If *n* is not given, the length of the input along the axis specified by *axis* is used.

**axis**

[int, optional] Axis over which to compute the FFT. If not given, the last axis is used.

**norm**

[{None, "ortho"}, optional] New in version 1.10.0.

Normalization mode (see `numpy.fft`). Default is None.

**Returns**

**out**

[complex ndarray] The truncated or zero-padded input, transformed along the axis indicated by *axis*, or the last one if *axis* is not specified.

**Raises**

**IndexError**

if *axes* is larger than the last axis of *a*.

**See also:**

**`numpy.fft`**

for definition of the DFT and conventions used.

**`ifft`**

The inverse of `fft`.

**`fft2`**

The two-dimensional FFT.

**`fftn`**

The *n*-dimensional FFT.

**`rfftn`**

The *n*-dimensional FFT of real input.

**`fftfreq`**

Frequency bins for given FFT parameters.

### Notes

FFT (Fast Fourier Transform) refers to a way the discrete Fourier Transform (DFT) can be calculated efficiently, by using symmetries in the calculated terms. The symmetry is highest when *n* is a power of 2, and the transform is therefore most efficient for these sizes.

The DFT is defined, with the conventions used in this implementation, in the documentation for the `numpy.fft` module.

### References

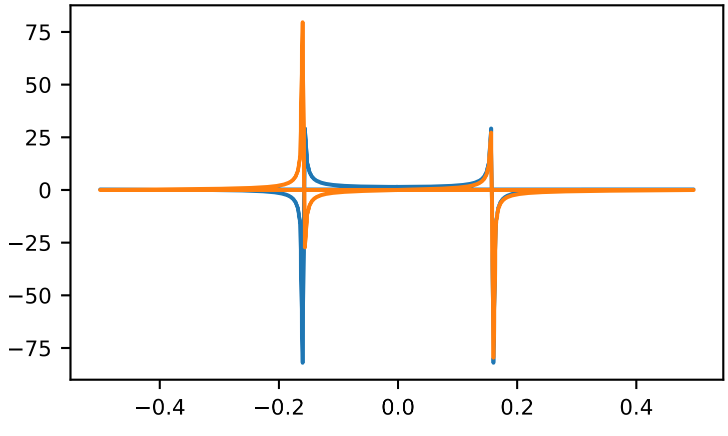
[CT]

### Examples

```
>>> np.fft.fft(np.exp(2j * np.pi * np.arange(8) / 8))
array([-2.33486982e-16+1.14423775e-17j,  8.00000000e+00-1.25557246e-15j,
        2.33486982e-16+2.33486982e-16j,  0.00000000e+00+1.22464680e-16j,
       -1.14423775e-17+2.33486982e-16j,  0.00000000e+00+5.20784380e-16j,
        1.14423775e-17+1.14423775e-17j,  0.00000000e+00+1.22464680e-16j])
```

In this example, real input has an FFT which is Hermitian, i.e., symmetric in the real part and anti-symmetric in the imaginary part, as described in the `numpy.fft` documentation:

```
>>> import matplotlib.pyplot as plt
>>> t = np.arange(256)
>>> sp = np.fft.fft(np.sin(t))
>>> freq = np.fft.fftfreq(t.shape[-1])
>>> plt.plot(freq, sp.real, freq, sp.imag)
[<matplotlib.lines.Line2D object at 0x...>, <matplotlib.lines.Line2D object at 0x.
↪..>]
>>> plt.show()
```



`numpy.fft.`**`ifft`**(*a*, *n=None*, *axis=-1*, *norm=None*)
Compute the one-dimensional inverse discrete Fourier Transform.

This function computes the inverse of the one-dimensional *n*-point discrete Fourier transform computed by *fft*. In other words, `ifft(fft(a)) == a` to within numerical accuracy. For a general description of the algorithm and definitions, see *numpy.fft*.

The input should be ordered in the same way as is returned by *fft*, i.e.,

- `a[0]` should contain the zero frequency term,
- `a[1:n//2]` should contain the positive-frequency terms,
- `a[n//2 + 1:]` should contain the negative-frequency terms, in increasing order starting from the most negative frequency.

For an even number of input points, `A[n//2]` represents the sum of the values at the positive and negative Nyquist frequencies, as the two are aliased together. See *numpy.fft* for details.

**Parameters**

**a**

[array_like] Input array, can be complex.

**n**

[int, optional] Length of the transformed axis of the output. If *n* is smaller than the length of the input, the input is cropped. If it is larger, the input is padded with zeros. If *n* is not given, the length of the input along the axis specified by *axis* is used. See notes about padding issues.

**axis**

[int, optional] Axis over which to compute the inverse DFT. If not given, the last axis is used.

**norm**

[{None, "ortho"}, optional] New in version 1.10.0.

Normalization mode (see *numpy.fft*). Default is None.

**Returns**

**out**

[complex ndarray] The truncated or zero-padded input, transformed along the axis indicated by *axis*, or the last one if *axis* is not specified.

**Raises**

**IndexError**

If *axes* is larger than the last axis of *a*.

**See also:**

***numpy.fft***

An introduction, with definitions and general explanations.

***fft***

The one-dimensional (forward) FFT, of which *ifft* is the inverse

***ifft2***

The two-dimensional inverse FFT.

***ifftn***

The n-dimensional inverse FFT.

### Notes

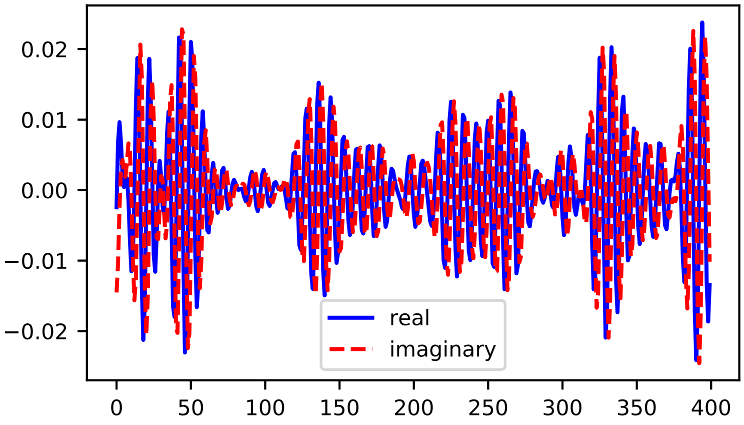
If the input parameter *n* is larger than the size of the input, the input is padded by appending zeros at the end. Even though this is the common approach, it might lead to surprising results. If a different padding is desired, it must be performed before calling *`ifft`*.

### Examples

```
>>> np.fft.ifft([0, 4, 0, 0])
array([ 1.+0.j,  0.+1.j, -1.+0.j,  0.-1.j]) # may vary
```

Create and plot a band-limited signal with random phases:

```
>>> import matplotlib.pyplot as plt
>>> t = np.arange(400)
>>> n = np.zeros((400,), dtype=complex)
>>> n[40:60] = np.exp(1j*np.random.uniform(0, 2*np.pi, (20,)))
>>> s = np.fft.ifft(n)
>>> plt.plot(t, s.real, 'b-', t, s.imag, 'r--')
[<matplotlib.lines.Line2D object at ...>, <matplotlib.lines.Line2D object at ...>]
>>> plt.legend(('real', 'imaginary'))
<matplotlib.legend.Legend object at ...>
>>> plt.show()
```



`numpy.fft.`**`fft2`**(*a*, *s=None*, *axes=(-2, -1)*, *norm=None*)

Compute the 2-dimensional discrete Fourier Transform

This function computes the *n*-dimensional discrete Fourier Transform over any axes in an *M*-dimensional array by means of the Fast Fourier Transform (FFT). By default, the transform is computed over the last two axes of the input array, i.e., a 2-dimensional FFT.

**Parameters**

**a**

[array_like] Input array, can be complex

**s**

[sequence of ints, optional] Shape (length of each transformed axis) of the output (`s[0]` refers to axis 0, `s[1]` to axis 1, etc.). This corresponds to `n` for `fft(x, n)`. Along each axis, if the given shape is smaller than that of the input, the input is cropped. If it is larger, the input is padded with zeros. if *s* is not given, the shape of the input along the axes specified by *axes* is used.

**axes**

[sequence of ints, optional] Axes over which to compute the FFT. If not given, the last two axes are used. A repeated index in *axes* means the transform over that axis is performed multiple times. A one-element sequence means that a one-dimensional FFT is performed.

**norm**

[{None, "ortho"}, optional] New in version 1.10.0.

Normalization mode (see *numpy.fft*). Default is None.

**Returns**

**out**

[complex ndarray] The truncated or zero-padded input, transformed along the axes indicated by *axes*, or the last two axes if *axes* is not given.

**Raises**

**ValueError**

If *s* and *axes* have different length, or *axes* not given and `len(s) != 2`.

**IndexError**

If an element of *axes* is larger than than the number of axes of *a*.

**See also:**

***numpy.fft***

Overall view of discrete Fourier transforms, with definitions and conventions used.

***ifft2***

The inverse two-dimensional FFT.

***fft***

The one-dimensional FFT.

***fftn***

The *n*-dimensional FFT.

***fftshift***

Shifts zero-frequency terms to the center of the array. For two-dimensional input, swaps first and third quadrants, and second and fourth quadrants.

## Notes

*`fft2`* is just *`fftn`* with a different default for *axes*.

The output, analogously to *`fft`*, contains the term for zero frequency in the low-order corner of the transformed axes, the positive frequency terms in the first half of these axes, the term for the Nyquist frequency in the middle of the axes and the negative frequency terms in the second half of the axes, in order of decreasingly negative frequency.

See *`fftn`* for details and a plotting example, and *`numpy.fft`* for definitions and conventions used.

## Examples

```
>>> a = np.mgrid[:5, :5][0]
>>> np.fft.fft2(a)
array([[ 50.  +0.j        ,   0.  +0.j        ,   0.  +0.j        , # may vary
          0.  +0.j        ,   0.  +0.j        ],
       [-12.5+17.20477401j,   0.  +0.j        ,   0.  +0.j        ,
          0.  +0.j        ,   0.  +0.j        ],
       [-12.5 +4.0614962j ,   0.  +0.j        ,   0.  +0.j        ,
          0.  +0.j        ,   0.  +0.j        ],
       [-12.5 -4.0614962j ,   0.  +0.j        ,   0.  +0.j        ,
          0.  +0.j        ,   0.  +0.j        ],
       [-12.5-17.20477401j,   0.  +0.j        ,   0.  +0.j        ,
          0.  +0.j        ,   0.  +0.j        ]])
```

`numpy.fft.`**`ifft2`**(*a*, *s=None*, *axes=(-2, -1)*, *norm=None*)

Compute the 2-dimensional inverse discrete Fourier Transform.

This function computes the inverse of the 2-dimensional discrete Fourier Transform over any number of axes in an M-dimensional array by means of the Fast Fourier Transform (FFT). In other words, `ifft2(fft2(a)) == a` to within numerical accuracy. By default, the inverse transform is computed over the last two axes of the input array.

The input, analogously to *`ifft`*, should be ordered in the same way as is returned by *`fft2`*, i.e. it should have the term for zero frequency in the low-order corner of the two axes, the positive frequency terms in the first half of these axes, the term for the Nyquist frequency in the middle of the axes and the negative frequency terms in the second half of both axes, in order of decreasingly negative frequency.

**Parameters**

**a**

[array_like] Input array, can be complex.

**s**

[sequence of ints, optional] Shape (length of each axis) of the output (`s[0]` refers to axis 0, `s[1]` to axis 1, etc.). This corresponds to *n* for `ifft(x, n)`. Along each axis, if the given shape is smaller than that of the input, the input is cropped. If it is larger, the input is padded with zeros. if *s* is not given, the shape of the input along the axes specified by *axes* is used. See notes for issue on *`ifft`* zero padding.

**axes**

[sequence of ints, optional] Axes over which to compute the FFT. If not given, the last two axes are used. A repeated index in *axes* means the transform over that axis is performed multiple times. A one-element sequence means that a one-dimensional FFT is performed.

**norm**

[{None, "ortho"}, optional] New in version 1.10.0.

Normalization mode (see `numpy.fft`). Default is None.

**Returns**

**out**

[complex ndarray] The truncated or zero-padded input, transformed along the axes indicated by *axes*, or the last two axes if *axes* is not given.

**Raises**

**ValueError**

If *s* and *axes* have different length, or *axes* not given and `len(s) != 2`.

**IndexError**

If an element of *axes* is larger than than the number of axes of *a*.

**See also:**

**`numpy.fft`**

Overall view of discrete Fourier transforms, with definitions and conventions used.

**`fft2`**

The forward 2-dimensional FFT, of which `ifft2` is the inverse.

**`ifftn`**

The inverse of the *n*-dimensional FFT.

**`fft`**

The one-dimensional FFT.

**`ifft`**

The one-dimensional inverse FFT.

## Notes

`ifft2` is just `ifftn` with a different default for *axes*.

See `ifftn` for details and a plotting example, and `numpy.fft` for definition and conventions used.

Zero-padding, analogously with `ifft`, is performed by appending zeros to the input along the specified dimension. Although this is the common approach, it might lead to surprising results. If another form of zero padding is desired, it must be performed before `ifft2` is called.

### Examples

```
>>> a = 4 * np.eye(4)
>>> np.fft.ifft2(a)
array([[1.+0.j,  0.+0.j,  0.+0.j,  0.+0.j], # may vary
       [0.+0.j,  0.+0.j,  0.+0.j,  1.+0.j],
       [0.+0.j,  0.+0.j,  1.+0.j,  0.+0.j],
       [0.+0.j,  1.+0.j,  0.+0.j,  0.+0.j]])
```

`numpy.fft.`**`fftn`**`(`*a*, *s=None*, *axes=None*, *norm=None*`)`

Compute the N-dimensional discrete Fourier Transform.

This function computes the *N*-dimensional discrete Fourier Transform over any number of axes in an *M*-dimensional array by means of the Fast Fourier Transform (FFT).

**Parameters**

**a**

[array_like] Input array, can be complex.

**s**

[sequence of ints, optional] Shape (length of each transformed axis) of the output (`s[0]` refers to axis 0, `s[1]` to axis 1, etc.). This corresponds to `n` for `fft(x, n)`. Along any axis, if the given shape is smaller than that of the input, the input is cropped. If it is larger, the input is padded with zeros. if *s* is not given, the shape of the input along the axes specified by *axes* is used.

**axes**

[sequence of ints, optional] Axes over which to compute the FFT. If not given, the last `len(s)` axes are used, or all axes if *s* is also not specified. Repeated indices in *axes* means that the transform over that axis is performed multiple times.

**norm**

[{None, "ortho"}, optional] New in version 1.10.0.

Normalization mode (see *numpy.fft*). Default is None.

**Returns**

**out**

[complex ndarray] The truncated or zero-padded input, transformed along the axes indicated by *axes*, or by a combination of *s* and *a*, as explained in the parameters section above.

**Raises**

**ValueError**

If *s* and *axes* have different length.

**IndexError**

If an element of *axes* is larger than than the number of axes of *a*.

**See also:**

***numpy.fft***

Overall view of discrete Fourier transforms, with definitions and conventions used.

***ifftn***

The inverse of *fftn*, the inverse *n*-dimensional FFT.

***fft***

The one-dimensional FFT, with definitions and conventions used.

***rfftn***

The *n*-dimensional FFT of real input.

***fft2***

The two-dimensional FFT.

***fftshift***

Shifts zero-frequency terms to centre of array

### Notes

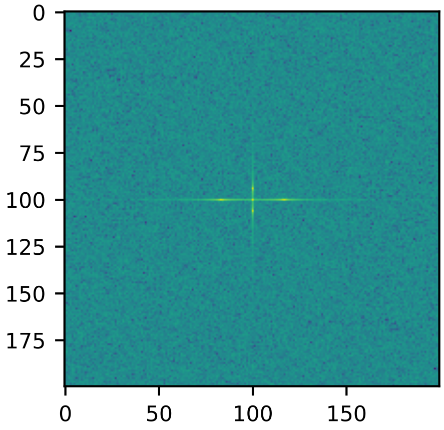
The output, analogously to *fft*, contains the term for zero frequency in the low-order corner of all axes, the positive frequency terms in the first half of all axes, the term for the Nyquist frequency in the middle of all axes and the negative frequency terms in the second half of all axes, in order of decreasingly negative frequency.

See *numpy.fft* for details, definitions and conventions used.

### Examples

```
>>> a = np.mgrid[:3, :3, :3][0]
>>> np.fft.fftn(a, axes=(1, 2))
array([[[ 0.+0.j,   0.+0.j,   0.+0.j], # may vary
        [ 0.+0.j,   0.+0.j,   0.+0.j],
        [ 0.+0.j,   0.+0.j,   0.+0.j]],
       [[ 9.+0.j,   0.+0.j,   0.+0.j],
        [ 0.+0.j,   0.+0.j,   0.+0.j],
        [ 0.+0.j,   0.+0.j,   0.+0.j]],
       [[18.+0.j,   0.+0.j,   0.+0.j],
        [ 0.+0.j,   0.+0.j,   0.+0.j],
        [ 0.+0.j,   0.+0.j,   0.+0.j]]])
>>> np.fft.fftn(a, (2, 2), axes=(0, 1))
array([[[ 2.+0.j,  2.+0.j,  2.+0.j], # may vary
        [ 0.+0.j,  0.+0.j,  0.+0.j]],
       [[-2.+0.j, -2.+0.j, -2.+0.j],
        [ 0.+0.j,  0.+0.j,  0.+0.j]]])
```

```
>>> import matplotlib.pyplot as plt
>>> [X, Y] = np.meshgrid(2 * np.pi * np.arange(200) / 12,
...                      2 * np.pi * np.arange(200) / 34)
>>> S = np.sin(X) + np.cos(Y) + np.random.uniform(0, 1, X.shape)
>>> FS = np.fft.fftn(S)
>>> plt.imshow(np.log(np.abs(np.fft.fftshift(FS))**2))
<matplotlib.image.AxesImage object at 0x...>
>>> plt.show()
```



`numpy.fft.`**`ifftn`**`(`*a*, *s=None*, *axes=None*, *norm=None*`)`

Compute the N-dimensional inverse discrete Fourier Transform.

This function computes the inverse of the N-dimensional discrete Fourier Transform over any number of axes in an M-dimensional array by means of the Fast Fourier Transform (FFT). In other words, `ifftn(fftn(a)) == a` to within numerical accuracy. For a description of the definitions and conventions used, see `numpy.fft`.

The input, analogously to `ifft`, should be ordered in the same way as is returned by `fftn`, i.e. it should have the term for zero frequency in all axes in the low-order corner, the positive frequency terms in the first half of all axes, the term for the Nyquist frequency in the middle of all axes and the negative frequency terms in the second half of all axes, in order of decreasingly negative frequency.

**Parameters**

**a**

[array_like] Input array, can be complex.

**s**

[sequence of ints, optional] Shape (length of each transformed axis) of the output (`s[0]` refers to axis 0, `s[1]` to axis 1, etc.). This corresponds to `n` for `ifft(x, n)`. Along any axis, if the given shape is smaller than that of the input, the input is cropped. If it is larger, the input is padded with zeros. if *s* is not given, the shape of the input along the axes specified by *axes* is used. See notes for issue on `ifft` zero padding.

**axes**

[sequence of ints, optional] Axes over which to compute the IFFT. If not given, the last `len(s)` axes are used, or all axes if *s* is also not specified. Repeated indices in *axes* means that the inverse transform over that axis is performed multiple times.

**norm**

[{None, "ortho"}, optional] New in version 1.10.0.

Normalization mode (see `numpy.fft`). Default is None.

**Returns**

**out**

[complex ndarray] The truncated or zero-padded input, transformed along the axes indicated by *axes*, or by a combination of *s* or *a*, as explained in the parameters section above.

**Raises**

**ValueError**

If *s* and *axes* have different length.

**IndexError**

If an element of *axes* is larger than than the number of axes of *a*.

**See also:**

**`numpy.fft`**

Overall view of discrete Fourier transforms, with definitions and conventions used.

**`fftn`**

The forward *n*-dimensional FFT, of which `ifftn` is the inverse.

**`ifft`**

The one-dimensional inverse FFT.

**`ifft2`**

The two-dimensional inverse FFT.

**`ifftshift`**

Undoes `fftshift`, shifts zero-frequency terms to beginning of array.

## Notes

See `numpy.fft` for definitions and conventions used.

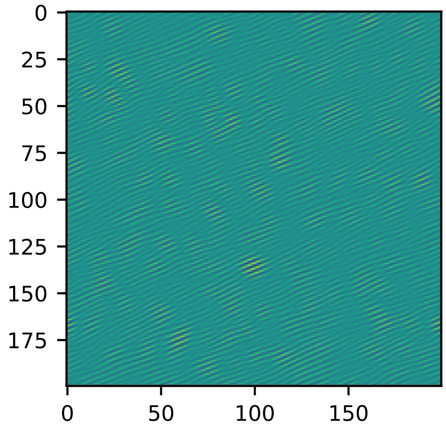
Zero-padding, analogously with `ifft`, is performed by appending zeros to the input along the specified dimension. Although this is the common approach, it might lead to surprising results. If another form of zero padding is desired, it must be performed before `ifftn` is called.

## Examples

```
>>> a = np.eye(4)
>>> np.fft.ifftn(np.fft.fftn(a, axes=(0,)), axes=(1,))
array([[1.+0.j,  0.+0.j,  0.+0.j,  0.+0.j], # may vary
       [0.+0.j,  1.+0.j,  0.+0.j,  0.+0.j],
       [0.+0.j,  0.+0.j,  1.+0.j,  0.+0.j],
       [0.+0.j,  0.+0.j,  0.+0.j,  1.+0.j]])
```

Create and plot an image with band-limited frequency content:

```
>>> import matplotlib.pyplot as plt
>>> n = np.zeros((200,200), dtype=complex)
>>> n[60:80, 20:40] = np.exp(1j*np.random.uniform(0, 2*np.pi, (20, 20)))
>>> im = np.fft.ifftn(n).real
>>> plt.imshow(im)
<matplotlib.image.AxesImage object at 0x...>
>>> plt.show()
```



## 4.11.2 Real FFTs

| | |
|---|---|
| *rfft*(a[, n, axis, norm]) | Compute the one-dimensional discrete Fourier Transform for real input. |
| *irfft*(a[, n, axis, norm]) | Compute the inverse of the n-point DFT for real input. |
| *rfft2*(a[, s, axes, norm]) | Compute the 2-dimensional FFT of a real array. |
| *irfft2*(a[, s, axes, norm]) | Compute the 2-dimensional inverse FFT of a real array. |
| *rfftn*(a[, s, axes, norm]) | Compute the N-dimensional discrete Fourier Transform for real input. |
| *irfftn*(a[, s, axes, norm]) | Compute the inverse of the N-dimensional FFT of real input. |

`numpy.fft.`**`rfft`**(*a*, *n=None*, *axis=-1*, *norm=None*)

Compute the one-dimensional discrete Fourier Transform for real input.

This function computes the one-dimensional *n*-point discrete Fourier Transform (DFT) of a real-valued array by means of an efficient algorithm called the Fast Fourier Transform (FFT).

**Parameters**

**a**

[array_like] Input array

**n**

[int, optional] Number of points along transformation axis in the input to use. If *n* is smaller than the length of the input, the input is cropped. If it is larger, the input is padded with zeros. If *n* is not given, the length of the input along the axis specified by *axis* is used.

**axis**

[int, optional] Axis over which to compute the FFT. If not given, the last axis is used.

**norm**

[{None, "ortho"}, optional] New in version 1.10.0.

Normalization mode (see `numpy.fft`). Default is None.

**Returns**

**out**

[complex ndarray] The truncated or zero-padded input, transformed along the axis indicated by *axis*, or the last one if *axis* is not specified. If *n* is even, the length of the transformed axis is `(n/2)+1`. If *n* is odd, the length is `(n+1)/2`.

**Raises**

**IndexError**

If *axis* is larger than the last axis of *a*.

**See also:**

**`numpy.fft`**

For definition of the DFT and conventions used.

**`irfft`**

The inverse of `rfft`.

**`fft`**

The one-dimensional FFT of general (complex) input.

**`fftn`**

The *n*-dimensional FFT.

**`rfftn`**

The *n*-dimensional FFT of real input.

## Notes

When the DFT is computed for purely real input, the output is Hermitian-symmetric, i.e. the negative frequency terms are just the complex conjugates of the corresponding positive-frequency terms, and the negative-frequency terms are therefore redundant. This function does not compute the negative frequency terms, and the length of the transformed axis of the output is therefore `n//2 + 1`.

When `A = rfft(a)` and fs is the sampling frequency, `A[0]` contains the zero-frequency term 0*fs, which is real due to Hermitian symmetry.

If *n* is even, `A[-1]` contains the term representing both positive and negative Nyquist frequency (+fs/2 and -fs/2), and must also be purely real. If *n* is odd, there is no term at fs/2; `A[-1]` contains the largest positive frequency (fs/2*(n-1)/n), and is complex in the general case.

If the input *a* contains an imaginary part, it is silently discarded.

### Examples

```
>>> np.fft.fft([0, 1, 0, 0])
array([ 1.+0.j,  0.-1.j, -1.+0.j,  0.+1.j]) # may vary
>>> np.fft.rfft([0, 1, 0, 0])
array([ 1.+0.j,  0.-1.j, -1.+0.j]) # may vary
```

Notice how the final element of the `fft` output is the complex conjugate of the second element, for real input. For `rfft`, this symmetry is exploited to compute only the non-negative frequency terms.

`numpy.fft.`**`irfft`**`(`*a*, *n=None*, *axis=-1*, *norm=None*`)`

Compute the inverse of the n-point DFT for real input.

This function computes the inverse of the one-dimensional *n*-point discrete Fourier Transform of real input computed by `rfft`. In other words, `irfft(rfft(a), len(a)) == a` to within numerical accuracy. (See Notes below for why `len(a)` is necessary here.)

The input is expected to be in the form returned by `rfft`, i.e. the real zero-frequency term followed by the complex positive frequency terms in order of increasing frequency. Since the discrete Fourier Transform of real input is Hermitian-symmetric, the negative frequency terms are taken to be the complex conjugates of the corresponding positive frequency terms.

**Parameters**

**a**

[array_like] The input array.

**n**

[int, optional] Length of the transformed axis of the output. For *n* output points, `n//2+1` input points are necessary. If the input is longer than this, it is cropped. If it is shorter than this, it is padded with zeros. If *n* is not given, it is taken to be `2*(m-1)` where `m` is the length of the input along the axis specified by *axis*.

**axis**

[int, optional] Axis over which to compute the inverse FFT. If not given, the last axis is used.

**norm**

[{None, "ortho"}, optional] New in version 1.10.0.

Normalization mode (see `numpy.fft`). Default is None.

**Returns**

**out**

[ndarray] The truncated or zero-padded input, transformed along the axis indicated by *axis*, or the last one if *axis* is not specified. The length of the transformed axis is *n*, or, if *n* is not given, `2*(m-1)` where `m` is the length of the transformed axis of the input. To get an odd number of output points, *n* must be specified.

**Raises**

**IndexError**

If *axis* is larger than the last axis of *a*.

**See also:**

***numpy.fft***

For definition of the DFT and conventions used.

***rfft***

The one-dimensional FFT of real input, of which *irfft* is inverse.

***fft***

The one-dimensional FFT.

***irfft2***

The inverse of the two-dimensional FFT of real input.

***irfftn***

The inverse of the *n*-dimensional FFT of real input.

## Notes

Returns the real valued *n*-point inverse discrete Fourier transform of *a*, where *a* contains the non-negative frequency terms of a Hermitian-symmetric sequence. *n* is the length of the result, not the input.

If you specify an *n* such that *a* must be zero-padded or truncated, the extra/removed values will be added/removed at high frequencies. One can thus resample a series to *m* points via Fourier interpolation by: `a_resamp = irfft(rfft(a), m)`.

The correct interpretation of the hermitian input depends on the length of the original data, as given by *n*. This is because each input shape could correspond to either an odd or even length signal. By default, *irfft* assumes an even output length which puts the last entry at the Nyquist frequency; aliasing with its symmetric counterpart. By Hermitian symmetry, the value is thus treated as purely real. To avoid losing information, the correct length of the real input **must** be given.

## Examples

```
>>> np.fft.ifft([1, -1j, -1, 1j])
array([0.+0.j,  1.+0.j,  0.+0.j,  0.+0.j]) # may vary
>>> np.fft.irfft([1, -1j, -1])
array([0.,  1.,  0.,  0.])
```

Notice how the last term in the input to the ordinary *ifft* is the complex conjugate of the second term, and the output has zero imaginary part everywhere. When calling *irfft*, the negative frequencies are not specified, and the output array is purely real.

`numpy.fft.`**`rfft2`**(*a*, *s=None*, *axes=(-2, -1)*, *norm=None*)

Compute the 2-dimensional FFT of a real array.

**Parameters**

**a**

[array] Input array, taken to be real.

**s**

[sequence of ints, optional] Shape of the FFT.

**axes**

[sequence of ints, optional] Axes over which to compute the FFT.

**norm**

[{None, "ortho"}, optional] New in version 1.10.0.

Normalization mode (see `numpy.fft`). Default is None.

**Returns**

**out**

[ndarray] The result of the real 2-D FFT.

**See also:**

**`rfftn`**

Compute the N-dimensional discrete Fourier Transform for real input.

## Notes

This is really just `rfftn` with different default behavior. For more details see `rfftn`.

`numpy.fft.`**`irfft2`**(*a*, *s=None*, *axes=(-2, -1)*, *norm=None*)

Compute the 2-dimensional inverse FFT of a real array.

**Parameters**

**a**

[array_like] The input array

**s**

[sequence of ints, optional] Shape of the real output to the inverse FFT.

**axes**

[sequence of ints, optional] The axes over which to compute the inverse fft. Default is the last two axes.

**norm**

[{None, "ortho"}, optional] New in version 1.10.0.

Normalization mode (see `numpy.fft`). Default is None.

**Returns**

**out**

[ndarray] The result of the inverse real 2-D FFT.

**See also:**

***irfftn***

Compute the inverse of the N-dimensional FFT of real input.

## Notes

This is really *irfftn* with different defaults. For more details see *irfftn*.

`numpy.fft.`**`rfftn`**(*a*, *s=None*, *axes=None*, *norm=None*)

Compute the N-dimensional discrete Fourier Transform for real input.

This function computes the N-dimensional discrete Fourier Transform over any number of axes in an M-dimensional real array by means of the Fast Fourier Transform (FFT). By default, all axes are transformed, with the real transform performed over the last axis, while the remaining transforms are complex.

**Parameters**

**a**

[array_like] Input array, taken to be real.

**s**

[sequence of ints, optional] Shape (length along each transformed axis) to use from the input. (`s[0]` refers to axis 0, `s[1]` to axis 1, etc.). The final element of *s* corresponds to *n* for `rfft(x, n)`, while for the remaining axes, it corresponds to *n* for `fft(x, n)`. Along any axis, if the given shape is smaller than that of the input, the input is cropped. If it is larger, the input is padded with zeros. if *s* is not given, the shape of the input along the axes specified by *axes* is used.

**axes**

[sequence of ints, optional] Axes over which to compute the FFT. If not given, the last `len(s)` axes are used, or all axes if *s* is also not specified.

**norm**

[{None, "ortho"}, optional] New in version 1.10.0.

Normalization mode (see *numpy.fft*). Default is None.

**Returns**

**out**

[complex ndarray] The truncated or zero-padded input, transformed along the axes indicated by *axes*, or by a combination of *s* and *a*, as explained in the parameters section above. The length of the last axis transformed will be `s[-1]//2+1`, while the remaining transformed axes will have lengths according to *s*, or unchanged from the input.

**Raises**

**ValueError**

If *s* and *axes* have different length.

**IndexError**

If an element of *axes* is larger than than the number of axes of *a*.

**See also:**

***irfftn***

The inverse of *rfftn*, i.e. the inverse of the n-dimensional FFT of real input.

***fft***

The one-dimensional FFT, with definitions and conventions used.

***rfft***

The one-dimensional FFT of real input.

***fftn***

The n-dimensional FFT.

***rfft2***

The two-dimensional FFT of real input.

## Notes

The transform for real input is performed over the last transformation axis, as by *rfft*, then the transform over the remaining axes is performed as by *fftn*. The order of the output is as for *rfft* for the final transformation axis, and as for *fftn* for the remaining transformation axes.

See *fft* for details, definitions and conventions used.

## Examples

```
>>> a = np.ones((2, 2, 2))
>>> np.fft.rfftn(a)
array([[[8.+0.j,  0.+0.j], # may vary
        [0.+0.j,  0.+0.j]],
       [[0.+0.j,  0.+0.j],
        [0.+0.j,  0.+0.j]]])
```

```
>>> np.fft.rfftn(a, axes=(2, 0))
array([[[4.+0.j,  0.+0.j], # may vary
        [4.+0.j,  0.+0.j]],
       [[0.+0.j,  0.+0.j],
        [0.+0.j,  0.+0.j]]])
```

`numpy.fft.`**`irfftn`**(*a*, *s=None*, *axes=None*, *norm=None*)

Compute the inverse of the N-dimensional FFT of real input.

This function computes the inverse of the N-dimensional discrete Fourier Transform for real input over any number of axes in an M-dimensional array by means of the Fast Fourier Transform (FFT). In other words, `irfftn(rfftn(a), a.shape) == a` to within numerical accuracy. (The `a.shape` is necessary like `len(a)` is for *irfft*, and for the same reason.)

The input should be ordered in the same way as is returned by *rfftn*, i.e. as for *irfft* for the final transformation axis, and as for *ifftn* along all the other axes.

**Parameters**

**a**

[array_like] Input array.

**s**

[sequence of ints, optional] Shape (length of each transformed axis) of the output (`s[0]` refers to axis 0, `s[1]` to axis 1, etc.). *s* is also the number of input points used along this axis, except for the last axis, where `s[-1]//2+1` points of the input are used. Along any axis, if the shape indicated by *s* is smaller than that of the input, the input is cropped. If it is larger, the input is padded with zeros. If *s* is not given, the shape of the input along the axes specified by axes is used. Except for the last axis which is taken to be `2*(m-1)` where `m` is the length of the input along that axis.

**axes**

[sequence of ints, optional] Axes over which to compute the inverse FFT. If not given, the last *len(s)* axes are used, or all axes if *s* is also not specified. Repeated indices in *axes* means that the inverse transform over that axis is performed multiple times.

**norm**

[{None, "ortho"}, optional] New in version 1.10.0.

Normalization mode (see `numpy.fft`). Default is None.

**Returns**

**out**

[ndarray] The truncated or zero-padded input, transformed along the axes indicated by *axes*, or by a combination of *s* or *a*, as explained in the parameters section above. The length of each transformed axis is as given by the corresponding element of *s*, or the length of the input in every axis except for the last one if *s* is not given. In the final transformed axis the length of the output when *s* is not given is `2*(m-1)` where `m` is the length of the final transformed axis of the input. To get an odd number of output points in the final axis, *s* must be specified.

**Raises**

**ValueError**

If *s* and *axes* have different length.

**IndexError**

If an element of *axes* is larger than than the number of axes of *a*.

**See also:**

**`rfftn`**

The forward n-dimensional FFT of real input, of which `ifftn` is the inverse.

**`fft`**

The one-dimensional FFT, with definitions and conventions used.

**`irfft`**

The inverse of the one-dimensional FFT of real input.

***irfft2***

The inverse of the two-dimensional FFT of real input.

### Notes

See *fft* for definitions and conventions used.

See *rfft* for definitions and conventions used for real input.

The correct interpretation of the hermitian input depends on the shape of the original data, as given by *s*. This is because each input shape could correspond to either an odd or even length signal. By default, *irfftn* assumes an even output length which puts the last entry at the Nyquist frequency; aliasing with its symmetric counterpart. When performing the final complex to real transform, the last value is thus treated as purely real. To avoid losing information, the correct shape of the real input **must** be given.

### Examples

```
>>> a = np.zeros((3, 2, 2))
>>> a[0, 0, 0] = 3 * 2 * 2
>>> np.fft.irfftn(a)
array([[[1.,  1.],
        [1.,  1.]],
       [[1.,  1.],
        [1.,  1.]],
       [[1.,  1.],
        [1.,  1.]]])
```

## 4.11.3 Hermitian FFTs

| | |
|---|---|
| *hfft*(a[, n, axis, norm]) | Compute the FFT of a signal that has Hermitian symmetry, i.e., a real spectrum. |
| *ihfft*(a[, n, axis, norm]) | Compute the inverse FFT of a signal that has Hermitian symmetry. |

`numpy.fft.`**`hfft`**(*a*, *n=None*, *axis=-1*, *norm=None*)

Compute the FFT of a signal that has Hermitian symmetry, i.e., a real spectrum.

**Parameters**

**a**

[array_like] The input array.

**n**

[int, optional] Length of the transformed axis of the output. For *n* output points, `n//2 + 1` input points are necessary. If the input is longer than this, it is cropped. If it is shorter than this, it is padded with zeros. If *n* is not given, it is taken to be `2*(m-1)` where `m` is the length of the input along the axis specified by *axis*.

**axis**

[int, optional] Axis over which to compute the FFT. If not given, the last axis is used.

**norm**

[{None, "ortho"}, optional] Normalization mode (see *numpy.fft*). Default is None.

New in version 1.10.0.

**Returns**

**out**

[ndarray] The truncated or zero-padded input, transformed along the axis indicated by *axis*, or the last one if *axis* is not specified. The length of the transformed axis is *n*, or, if *n* is not given, `2*m - 2` where `m` is the length of the transformed axis of the input. To get an odd number of output points, *n* must be specified, for instance as `2*m - 1` in the typical case,

**Raises**

**IndexError**

If *axis* is larger than the last axis of *a*.

**See also:**

***rfft***

Compute the one-dimensional FFT for real input.

***ihfft***

The inverse of *hfft*.

## Notes

*hfft*/*ihfft* are a pair analogous to *rfft*/*irfft*, but for the opposite case: here the signal has Hermitian symmetry in the time domain and is real in the frequency domain. So here it's *hfft* for which you must supply the length of the result if it is to be odd.

- even: `ihfft(hfft(a, 2*len(a) - 2)) == a`, within roundoff error,
- odd: `ihfft(hfft(a, 2*len(a) - 1)) == a`, within roundoff error.

The correct interpretation of the hermitian input depends on the length of the original data, as given by *n*. This is because each input shape could correspond to either an odd or even length signal. By default, *hfft* assumes an even output length which puts the last entry at the Nyquist frequency; aliasing with its symmetric counterpart. By Hermitian symmetry, the value is thus treated as purely real. To avoid losing information, the shape of the full signal **must** be given.

## Examples

```
>>> signal = np.array([1, 2, 3, 4, 3, 2])
>>> np.fft.fft(signal)
array([15.+0.j,  -4.+0.j,   0.+0.j,  -1.-0.j,   0.+0.j,  -4.+0.j]) # may vary
>>> np.fft.hfft(signal[:4]) # Input first half of signal
array([15.,  -4.,   0.,  -1.,   0.,  -4.])
>>> np.fft.hfft(signal, 6)  # Input entire signal and truncate
array([15.,  -4.,   0.,  -1.,   0.,  -4.])
```

```
>>> signal = np.array([[1, 1.j], [-1.j, 2]])
>>> np.conj(signal.T) - signal   # check Hermitian symmetry
array([[ 0.-0.j,  -0.+0.j], # may vary
       [ 0.+0.j,  0.-0.j]])
>>> freq_spectrum = np.fft.hfft(signal)
>>> freq_spectrum
array([[ 1.,  1.],
       [ 2., -2.]])
```

`numpy.fft.`**`ihfft`**(*a*, *n=None*, *axis=-1*, *norm=None*)

Compute the inverse FFT of a signal that has Hermitian symmetry.

**Parameters**

**a**

[array_like] Input array.

**n**

[int, optional] Length of the inverse FFT, the number of points along transformation axis in the input to use. If *n* is smaller than the length of the input, the input is cropped. If it is larger, the input is padded with zeros. If *n* is not given, the length of the input along the axis specified by *axis* is used.

**axis**

[int, optional] Axis over which to compute the inverse FFT. If not given, the last axis is used.

**norm**

[{None, "ortho"}, optional] Normalization mode (see *`numpy.fft`*). Default is None.

New in version 1.10.0.

**Returns**

**out**

[complex ndarray] The truncated or zero-padded input, transformed along the axis indicated by *axis*, or the last one if *axis* is not specified. The length of the transformed axis is `n//2 + 1`.

**See also:**

*`hfft`*, *`irfft`*

## Notes

*`hfft`*/*`ihfft`* are a pair analogous to *`rfft`*/*`irfft`*, but for the opposite case: here the signal has Hermitian symmetry in the time domain and is real in the frequency domain. So here it's *`hfft`* for which you must supply the length of the result if it is to be odd:

- even: `ihfft(hfft(a, 2*len(a) - 2)) == a`, within roundoff error,
- odd: `ihfft(hfft(a, 2*len(a) - 1)) == a`, within roundoff error.

### Examples

```
>>> spectrum = np.array([ 15, -4, 0, -1, 0, -4])
>>> np.fft.ifft(spectrum)
array([1.+0.j,  2.+0.j,  3.+0.j,  4.+0.j,  3.+0.j,  2.+0.j]) # may vary
>>> np.fft.ihfft(spectrum)
array([ 1.-0.j,  2.-0.j,  3.-0.j,  4.-0.j]) # may vary
```

## 4.11.4 Helper routines

| | |
|---|---|
| *fftfreq*(n[, d]) | Return the Discrete Fourier Transform sample frequencies. |
| *rfftfreq*(n[, d]) | Return the Discrete Fourier Transform sample frequencies (for usage with rfft, irfft). |
| *fftshift*(x[, axes]) | Shift the zero-frequency component to the center of the spectrum. |
| *ifftshift*(x[, axes]) | The inverse of *fftshift*. |

`numpy.fft.`**`fftfreq`**`(`*n, d=1.0*`)`

Return the Discrete Fourier Transform sample frequencies.

The returned float array *f* contains the frequency bin centers in cycles per unit of the sample spacing (with zero at the start). For instance, if the sample spacing is in seconds, then the frequency unit is cycles/second.

Given a window length *n* and a sample spacing *d*:

```
f = [0, 1, ...,   n/2-1,     -n/2, ..., -1] / (d*n)   if n is even
f = [0, 1, ..., (n-1)/2, -(n-1)/2, ..., -1] / (d*n)   if n is odd
```

**Parameters**

**n**

[int] Window length.

**d**

[scalar, optional] Sample spacing (inverse of the sampling rate). Defaults to 1.

**Returns**

**f**

[ndarray] Array of length *n* containing the sample frequencies.

### Examples

```
>>> signal = np.array([-2, 8, 6, 4, 1, 0, 3, 5], dtype=float)
>>> fourier = np.fft.fft(signal)
>>> n = signal.size
>>> timestep = 0.1
>>> freq = np.fft.fftfreq(n, d=timestep)
>>> freq
array([ 0.  ,  1.25,  2.5 , ..., -3.75, -2.5 , -1.25])
```

`numpy.fft.`**`rfftfreq`**(*n*, *d=1.0*)

Return the Discrete Fourier Transform sample frequencies (for usage with rfft, irfft).

The returned float array *f* contains the frequency bin centers in cycles per unit of the sample spacing (with zero at the start). For instance, if the sample spacing is in seconds, then the frequency unit is cycles/second.

Given a window length *n* and a sample spacing *d*:

```
f = [0, 1, ...,     n/2-1,     n/2] / (d*n)   if n is even
f = [0, 1, ..., (n-1)/2-1, (n-1)/2] / (d*n)   if n is odd
```

Unlike *`fftfreq`* (but like `scipy.fftpack.rfftfreq`) the Nyquist frequency component is considered to be positive.

**Parameters**

**n**

[int] Window length.

**d**

[scalar, optional] Sample spacing (inverse of the sampling rate). Defaults to 1.

**Returns**

**f**

[ndarray] Array of length `n//2 + 1` containing the sample frequencies.

### Examples

```
>>> signal = np.array([-2, 8, 6, 4, 1, 0, 3, 5, -3, 4], dtype=float)
>>> fourier = np.fft.rfft(signal)
>>> n = signal.size
>>> sample_rate = 100
>>> freq = np.fft.fftfreq(n, d=1./sample_rate)
>>> freq
array([  0.,  10.,  20., ..., -30., -20., -10.])
>>> freq = np.fft.rfftfreq(n, d=1./sample_rate)
>>> freq
array([  0.,  10.,  20.,  30.,  40.,  50.])
```

`numpy.fft.`**`fftshift`**(*x*, *axes=None*)

Shift the zero-frequency component to the center of the spectrum.

This function swaps half-spaces for all axes listed (defaults to all). Note that `y[0]` is the Nyquist component only if `len(x)` is even.

**Parameters**

**x**

[array_like] Input array.

**axes**

[int or shape tuple, optional] Axes over which to shift. Default is None, which shifts all axes.

**Returns**

**y**

[ndarray] The shifted array.

**See also:**

***ifftshift***

The inverse of *fftshift*.

## Examples

```
>>> freqs = np.fft.fftfreq(10, 0.1)
>>> freqs
array([ 0.,  1.,  2., ..., -3., -2., -1.])
>>> np.fft.fftshift(freqs)
array([-5., -4., -3., -2., -1.,  0.,  1.,  2.,  3.,  4.])
```

Shift the zero-frequency component only along the second axis:

```
>>> freqs = np.fft.fftfreq(9, d=1./9).reshape(3, 3)
>>> freqs
array([[ 0.,  1.,  2.],
       [ 3.,  4., -4.],
       [-3., -2., -1.]])
>>> np.fft.fftshift(freqs, axes=(1,))
array([[ 2.,  0.,  1.],
       [-4.,  3.,  4.],
       [-1., -3., -2.]])
```

`numpy.fft.`**`ifftshift`**(*x*, *axes=None*)

The inverse of *fftshift*. Although identical for even-length *x*, the functions differ by one sample for odd-length *x*.

**Parameters**

**x**

[array_like] Input array.

**axes**

[int or shape tuple, optional] Axes over which to calculate. Defaults to None, which shifts all axes.

**Returns**

**y**

[ndarray] The shifted array.

**See also:**

***fftshift***

Shift zero-frequency component to the center of the spectrum.

### Examples

```
>>> freqs = np.fft.fftfreq(9, d=1./9).reshape(3, 3)
>>> freqs
array([[ 0.,  1.,  2.],
       [ 3.,  4., -4.],
       [-3., -2., -1.]])
>>> np.fft.ifftshift(np.fft.fftshift(freqs))
array([[ 0.,  1.,  2.],
       [ 3.,  4., -4.],
       [-3., -2., -1.]])
```

## 4.11.5 Background information

Fourier analysis is fundamentally a method for expressing a function as a sum of periodic components, and for recovering the function from those components. When both the function and its Fourier transform are replaced with discretized counterparts, it is called the discrete Fourier transform (DFT). The DFT has become a mainstay of numerical computing in part because of a very fast algorithm for computing it, called the Fast Fourier Transform (FFT), which was known to Gauss (1805) and was brought to light in its current form by Cooley and Tukey [CT]. Press et al. [NR] provide an accessible introduction to Fourier analysis and its applications.

Because the discrete Fourier transform separates its input into components that contribute at discrete frequencies, it has a great number of applications in digital signal processing, e.g., for filtering, and in this context the discretized input to the transform is customarily referred to as a *signal*, which exists in the *time domain*. The output is called a *spectrum* or *transform* and exists in the *frequency domain*.

## 4.11.6 Implementation details

There are many ways to define the DFT, varying in the sign of the exponent, normalization, etc. In this implementation, the DFT is defined as

$$A_k = \sum_{m=0}^{n-1} a_m \exp\left\{-2\pi i\frac{mk}{n}\right\} \qquad k = 0,\ldots,n-1.$$

The DFT is in general defined for complex inputs and outputs, and a single-frequency component at linear frequency $f$ is represented by a complex exponential $a_m = \exp\{2\pi i\,fm\Delta t\}$, where $\Delta t$ is the sampling interval.

The values in the result follow so-called "standard" order: If `A = fft(a, n)`, then `A[0]` contains the zero-frequency term (the sum of the signal), which is always purely real for real inputs. Then `A[1:n/2]` contains the positive-frequency terms, and `A[n/2+1:]` contains the negative-frequency terms, in order of decreasingly negative frequency. For an even number of input points, `A[n/2]` represents both positive and negative Nyquist frequency, and is also purely real for real input. For an odd number of input points, `A[(n-1)/2]` contains the largest positive frequency, while `A[(n+1)/2]` contains the largest negative frequency. The routine `np.fft.fftfreq(n)` returns an array giving the frequencies of corresponding elements in the output. The routine `np.fft.fftshift(A)` shifts transforms and their frequencies to put the zero-frequency components in the middle, and `np.fft.ifftshift(A)` undoes that shift.

When the input *a* is a time-domain signal and `A = fft(a)`, `np.abs(A)` is its amplitude spectrum and `np.abs(A)**2` is its power spectrum. The phase spectrum is obtained by `np.angle(A)`.

The inverse DFT is defined as

$$a_m = \frac{1}{n}\sum_{k=0}^{n-1} A_k \exp\left\{2\pi i\frac{mk}{n}\right\} \qquad m = 0,\ldots,n-1.$$

It differs from the forward transform by the sign of the exponential argument and the default normalization by $1/n$.

### 4.11.7 Type Promotion

*`numpy.fft`* promotes `float32` and `complex64` arrays to `float64` and `complex128` arrays respectively. For an FFT implementation that does not promote input arrays, see `scipy.fftpack`.

### 4.11.8 Normalization

The default normalization has the direct transforms unscaled and the inverse transforms are scaled by $1/n$. It is possible to obtain unitary transforms by setting the keyword argument `norm` to `"ortho"` (default is *None*) so that both direct and inverse transforms will be scaled by $1/\sqrt{n}$.

### 4.11.9 Real and Hermitian transforms

When the input is purely real, its transform is Hermitian, i.e., the component at frequency $f_k$ is the complex conjugate of the component at frequency $-f_k$, which means that for real inputs there is no information in the negative frequency components that is not already available from the positive frequency components. The family of *`rfft`* functions is designed to operate on real inputs, and exploits this symmetry by computing only the positive frequency components, up to and including the Nyquist frequency. Thus, `n` input points produce `n/2+1` complex output points. The inverses of this family assumes the same symmetry of its input, and for an output of `n` points uses `n/2+1` input points.

Correspondingly, when the spectrum is purely real, the signal is Hermitian. The *`hfft`* family of functions exploits this symmetry by using `n/2+1` complex points in the input (time) domain for `n` real points in the frequency domain.

In higher dimensions, FFTs are used, e.g., for image analysis and filtering. The computational efficiency of the FFT means that it can also be a faster way to compute large convolutions, using the property that a convolution in the time domain is equivalent to a point-by-point multiplication in the frequency domain.

### 4.11.10 Higher dimensions

In two dimensions, the DFT is defined as

$$A_{kl} = \sum_{m=0}^{M-1}\sum_{n=0}^{N-1} a_{mn}\exp\left\{-2\pi i\left(\frac{mk}{M}+\frac{nl}{N}\right)\right\} \qquad k = 0,\ldots,M-1;\quad l = 0,\ldots,N-1,$$

which extends in the obvious way to higher dimensions, and the inverses in higher dimensions also extend in the same way.

### 4.11.11 References

### 4.11.12 Examples

For examples, see the various functions.

## 4.12 Financial functions

### 4.12.1 Simple financial functions

| | |
|---|---|
| `fv`(rate, nper, pmt, pv[, when]) | Compute the future value. |
| `pv`(rate, nper, pmt[, fv, when]) | Compute the present value. |
| `npv`(rate, values) | Returns the NPV (Net Present Value) of a cash flow series. |
| `pmt`(rate, nper, pv[, fv, when]) | Compute the payment against loan principal plus interest. |
| `ppmt`(rate, per, nper, pv[, fv, when]) | Compute the payment against loan principal. |
| `ipmt`(rate, per, nper, pv[, fv, when]) | Compute the interest portion of a payment. |
| `irr`(values) | Return the Internal Rate of Return (IRR). |
| `mirr`(values, finance_rate, reinvest_rate) | Modified internal rate of return. |
| `nper`(rate, pmt, pv[, fv, when]) | Compute the number of periodic payments. |
| `rate`(nper, pmt, pv, fv[, when, guess, tol, …]) | Compute the rate of interest per period. |

`numpy.`**`fv`**(*rate*, *nper*, *pmt*, *pv*, *when='end'*)

Compute the future value.

Deprecated since version 1.18: `fv` is deprecated; for details, see NEP 32 [1]. Use the corresponding function in the numpy-financial library, https://pypi.org/project/numpy-financial.

**Given:**

- a present value, `pv`
- an interest `rate` compounded once per period, of which there are
- `nper` total
- a (fixed) payment, `pmt`, paid either
- at the beginning (*when* = {'begin', 1}) or the end (*when* = {'end', 0}) of each period

**Return:**

the value at the end of the `nper` periods

**Parameters**

**rate**

[scalar or array_like of shape(M, )] Rate of interest as decimal (not per cent) per period

**nper**

[scalar or array_like of shape(M, )] Number of compounding periods

**pmt**

[scalar or array_like of shape(M, )] Payment

**pv**

[scalar or array_like of shape(M, )] Present value

**when**

[{{'begin', 1}, {'end', 0}}, {string, int}, optional] When payments are due ('begin' (1) or 'end' (0)). Defaults to {'end', 0}.

**Returns**

**out**

[ndarray] Future values. If all input is scalar, returns a scalar float. If any input is array_like, returns future values for each input element. If multiple inputs are array_like, they all must have the same shape.

## Notes

The future value is computed by solving the equation:

```
fv +
pv*(1+rate)**nper +
pmt*(1 + rate*when)/rate*((1 + rate)**nper - 1) == 0
```

or, when `rate == 0`:

```
fv + pv + pmt * nper == 0
```

## References

[1], [2]

## Examples

What is the future value after 10 years of saving $100 now, with an additional monthly savings of $100. Assume the interest rate is 5% (annually) compounded monthly?

```
>>> np.fv(0.05/12, 10*12, -100, -100)
15692.928894335748
```

By convention, the negative sign represents cash flow out (i.e. money not available today). Thus, saving $100 a month at 5% annual interest leads to $15,692.93 available to spend in 10 years.

If any input is array_like, returns an array of equal shape. Let's compare different interest rates from the example above.

```
>>> a = np.array((0.05, 0.06, 0.07))/12
>>> np.fv(a, 10*12, -100, -100)
array([ 15692.92889434,  16569.87435405,  17509.44688102]) # may vary
```

`numpy.pv`(*rate*, *nper*, *pmt*, *fv=0*, *when='end'*)

Compute the present value.

Deprecated since version 1.18: `pv` is deprecated; for details, see NEP 32 [1]. Use the corresponding function in the numpy-financial library, https://pypi.org/project/numpy-financial.

**Given:**

- a future value, `fv`
- an interest `rate` compounded once per period, of which there are
- `nper` total
- a (fixed) payment, `pmt`, paid either
- at the beginning (*when* = {'begin', 1}) or the end (*when* = {'end', 0}) of each period

**Return:**

the value now

**Parameters**

**rate**

[array_like] Rate of interest (per period)

**nper**

[array_like] Number of compounding periods

**pmt**

[array_like] Payment

**fv**

[array_like, optional] Future value

**when**

[{{'begin', 1}, {'end', 0}}, {string, int}, optional] When payments are due ('begin' (1) or 'end' (0))

**Returns**

**out**

[ndarray, float] Present value of a series of payments or investments.

## Notes

The present value is computed by solving the equation:

```
fv +
pv*(1 + rate)**nper +
pmt*(1 + rate*when)/rate*((1 + rate)**nper - 1) = 0
```

or, when `rate = 0`:

```
fv + pv + pmt * nper = 0
```

for *pv*, which is then returned.

### References

[1], [2]

### Examples

What is the present value (e.g., the initial investment) of an investment that needs to total $15692.93 after 10 years of saving $100 every month? Assume the interest rate is 5% (annually) compounded monthly.

```
>>> np.pv(0.05/12, 10*12, -100, 15692.93)
-100.00067131625819
```

By convention, the negative sign represents cash flow out (i.e., money not available today). Thus, to end up with $15,692.93 in 10 years saving $100 a month at 5% annual interest, one's initial deposit should also be $100.

If any input is array_like, `pv` returns an array of equal shape. Let's compare different interest rates in the example above:

```
>>> a = np.array((0.05, 0.04, 0.03))/12
>>> np.pv(a, 10*12, -100, 15692.93)
array([ -100.00067132,  -649.26771385, -1273.78633713]) # may vary
```

So, to end up with the same $15692.93 under the same $100 per month "savings plan," for annual interest rates of 4% and 3%, one would need initial investments of $649.27 and $1273.79, respectively.

`numpy.`**`npv`**`(`*rate*, *values*`)`

Returns the NPV (Net Present Value) of a cash flow series.

Deprecated since version 1.18: *npv* is deprecated; for details, see NEP 32 [1]. Use the corresponding function in the numpy-financial library, https://pypi.org/project/numpy-financial.

**Parameters**

**rate**

[scalar] The discount rate.

**values**

[array_like, shape(M, )] The values of the time series of cash flows. The (fixed) time interval between cash flow "events" must be the same as that for which *rate* is given (i.e., if *rate* is per year, then precisely a year is understood to elapse between each cash flow event). By convention, investments or "deposits" are negative, income or "withdrawals" are positive; *values* must begin with the initial investment, thus *values[0]* will typically be negative.

**Returns**

**out**

[float] The NPV of the input cash flow series *values* at the discount *rate*.

> **Warning:** `npv` considers a series of cashflows starting in the present (t = 0). NPV can also be defined with a series of future cashflows, paid at the end, rather than the start, of each period. If future cashflows are used, the first cashflow *values[0]* must be zeroed and added to the net present value of the future cashflows. This is demonstrated in the examples.

### Notes

Returns the result of: [2]

$$\sum_{t=0}^{M-1} \frac{values_t}{(1 + rate)^t}$$

### References

[1], [2]

### Examples

Consider a potential project with an initial investment of $40 000 and projected cashflows of $5 000, $8 000, $12 000 and $30 000 at the end of each period discounted at a rate of 8% per period. To find the project's net present value:

```
>>> rate, cashflows = 0.08, [-40_000, 5_000, 8_000, 12_000, 30_000]
>>> np.npv(rate, cashflows).round(5)
3065.22267
```

It may be preferable to split the projected cashflow into an initial investment and expected future cashflows. In this case, the value of the initial cashflow is zero and the initial investment is later added to the future cashflows net present value:

```
>>> initial_cashflow = cashflows[0]
>>> cashflows[0] = 0
>>> np.round(np.npv(rate, cashflows) + initial_cashflow, 5)
3065.22267
```

`numpy.`**`pmt`**(*rate*, *nper*, *pv*, *fv=0*, *when='end'*)

Compute the payment against loan principal plus interest.

Deprecated since version 1.18: `pmt` is deprecated; for details, see NEP 32 [1]. Use the corresponding function in the numpy-financial library, https://pypi.org/project/numpy-financial.

**Given:**

- a present value, `pv` (e.g., an amount borrowed)
- a future value, `fv` (e.g., 0)
- an interest `rate` compounded once per period, of which there are
- `nper` total
- and (optional) specification of whether payment is made at the beginning (*when* = {'begin', 1}) or the end (*when* = {'end', 0}) of each period

**Return:**

the (fixed) periodic payment.

**Parameters**

**rate**

[array_like] Rate of interest (per period)

**nper**

[array_like] Number of compounding periods

**pv**

[array_like] Present value

**fv**

[array_like, optional] Future value (default = 0)

**when**

[{{'begin', 1}, {'end', 0}}, {string, int}] When payments are due ('begin' (1) or 'end' (0))

**Returns**

**out**

[ndarray] Payment against loan plus interest. If all input is scalar, returns a scalar float. If any input is array_like, returns payment for each input element. If multiple inputs are array_like, they all must have the same shape.

## Notes

The payment is computed by solving the equation:

```
fv +
pv*(1 + rate)**nper +
pmt*(1 + rate*when)/rate*((1 + rate)**nper - 1) == 0
```

or, when `rate == 0`:

```
fv + pv + pmt * nper == 0
```

for `pmt`.

Note that computing a monthly mortgage payment is only one use for this function. For example, pmt returns the periodic deposit one must make to achieve a specified future balance given an initial deposit, a fixed, periodically compounded interest rate, and the total number of periods.

### References

[1], [2]

### Examples

What is the monthly payment needed to pay off a $200,000 loan in 15 years at an annual interest rate of 7.5%?

```
>>> np.pmt(0.075/12, 12*15, 200000)
-1854.0247200054619
```

In order to pay-off (i.e., have a future-value of 0) the $200,000 obtained today, a monthly payment of $1,854.02 would be required. Note that this example illustrates usage of `fv` having a default value of 0.

`numpy.`**`ppmt`**(*rate*, *per*, *nper*, *pv*, *fv=0*, *when='end'*)

Compute the payment against loan principal.

Deprecated since version 1.18: `ppmt` is deprecated; for details, see NEP 32 [1]. Use the corresponding function in the numpy-financial library, https://pypi.org/project/numpy-financial.

**Parameters**

**rate**

[array_like] Rate of interest (per period)

**per**

[array_like, int] Amount paid against the loan changes. The *per* is the period of interest.

**nper**

[array_like] Number of compounding periods

**pv**

[array_like] Present value

**fv**

[array_like, optional] Future value

**when**

[{{'begin', 1}, {'end', 0}}, {string, int}] When payments are due ('begin' (1) or 'end' (0))

**See also:**

`pmt`, `pv`, `ipmt`

### References

[1]

`numpy.`**`ipmt`**(*rate*, *per*, *nper*, *pv*, *fv=0*, *when='end'*)

Compute the interest portion of a payment.

Deprecated since version 1.18: `ipmt` is deprecated; for details, see NEP 32 [1]. Use the corresponding function in the numpy-financial library, https://pypi.org/project/numpy-financial.

**Parameters**

**rate**

[scalar or array_like of shape(M, )] Rate of interest as decimal (not per cent) per period

**per**

[scalar or array_like of shape(M, )] Interest paid against the loan changes during the life or the loan. The *per* is the payment period to calculate the interest amount.

**nper**

[scalar or array_like of shape(M, )] Number of compounding periods

**pv**

[scalar or array_like of shape(M, )] Present value

**fv**

[scalar or array_like of shape(M, ), optional] Future value

**when**

[{{'begin', 1}, {'end', 0}}, {string, int}, optional] When payments are due ('begin' (1) or 'end' (0)). Defaults to {'end', 0}.

**Returns**

**out**

[ndarray] Interest portion of payment. If all input is scalar, returns a scalar float. If any input is array_like, returns interest payment for each input element. If multiple inputs are array_like, they all must have the same shape.

**See also:**

*ppmt*, *pmt*, *pv*

### Notes

The total payment is made up of payment against principal plus interest.

`pmt = ppmt + ipmt`

### References

[1]

### Examples

What is the amortization schedule for a 1 year loan of $2500 at 8.24% interest per year compounded monthly?

```
>>> principal = 2500.00
```

The 'per' variable represents the periods of the loan. Remember that financial equations start the period count at 1!

```
>>> per = np.arange(1*12) + 1
>>> ipmt = np.ipmt(0.0824/12, per, 1*12, principal)
>>> ppmt = np.ppmt(0.0824/12, per, 1*12, principal)
```

Each element of the sum of the 'ipmt' and 'ppmt' arrays should equal 'pmt'.

```
>>> pmt = np.pmt(0.0824/12, 1*12, principal)
>>> np.allclose(ipmt + ppmt, pmt)
True
```

```
>>> fmt = '{0:2d} {1:8.2f} {2:8.2f} {3:8.2f}'
>>> for payment in per:
...     index = payment - 1
...     principal = principal + ppmt[index]
...     print(fmt.format(payment, ppmt[index], ipmt[index], principal))
 1  -200.58   -17.17  2299.42
 2  -201.96   -15.79  2097.46
 3  -203.35   -14.40  1894.11
 4  -204.74   -13.01  1689.37
 5  -206.15   -11.60  1483.22
 6  -207.56   -10.18  1275.66
 7  -208.99    -8.76  1066.67
 8  -210.42    -7.32   856.25
 9  -211.87    -5.88   644.38
10  -213.32    -4.42   431.05
11  -214.79    -2.96   216.26
12  -216.26    -1.49    -0.00
```

```
>>> interestpd = np.sum(ipmt)
>>> np.round(interestpd, 2)
-112.98
```

`numpy.`**`irr`**`(`*values*`)`

Return the Internal Rate of Return (IRR).

Deprecated since version 1.18: *`irr`* is deprecated; for details, see NEP 32 [1]. Use the corresponding function in the numpy-financial library, https://pypi.org/project/numpy-financial.

This is the "average" periodically compounded rate of return that gives a net present value of 0.0; for a more complete explanation, see Notes below.

`decimal.Decimal` type is not supported.

**Parameters**

**values**

[array_like, shape(N,)] Input cash flows per time period. By convention, net "deposits" are negative and net "withdrawals" are positive. Thus, for example, at least the first element of *values*, which represents the initial investment, will typically be negative.

**Returns**

**out**

[float] Internal Rate of Return for periodic input values.

### Notes

The IRR is perhaps best understood through an example (illustrated using np.irr in the Examples section below). Suppose one invests 100 units and then makes the following withdrawals at regular (fixed) intervals: 39, 59, 55, 20. Assuming the ending value is 0, one's 100 unit investment yields 173 units; however, due to the combination of compounding and the periodic withdrawals, the "average" rate of return is neither simply 0.73/4 nor (1.73)^0.25-1. Rather, it is the solution (for $r$) of the equation:

$$-100 + \frac{39}{1+r} + \frac{59}{(1+r)^2} + \frac{55}{(1+r)^3} + \frac{20}{(1+r)^4} = 0$$

In general, for *values* $= [v_0, v_1, ...v_M]$, irr is the solution of the equation: [2]

$$\sum_{t=0}^{M} \frac{v_t}{(1+irr)^t} = 0$$

### References

[1], [2]

### Examples

```
>>> round(np.irr([-100, 39, 59, 55, 20]), 5)
0.28095
>>> round(np.irr([-100, 0, 0, 74]), 5)
-0.0955
>>> round(np.irr([-100, 100, 0, -7]), 5)
-0.0833
>>> round(np.irr([-100, 100, 0, 7]), 5)
0.06206
>>> round(np.irr([-5, 10.5, 1, -8, 1]), 5)
0.0886
```

`numpy.`**`mirr`**(*values*, *finance_rate*, *reinvest_rate*)

Modified internal rate of return.

Deprecated since version 1.18: *`mirr`* is deprecated; for details, see NEP 32 [1]. Use the corresponding function in the numpy-financial library, https://pypi.org/project/numpy-financial.

**Parameters**

**values**

[array_like] Cash flows (must contain at least one positive and one negative value) or nan is returned. The first value is considered a sunk cost at time zero.

**finance_rate**

[scalar] Interest rate paid on the cash flows

**reinvest_rate**

[scalar] Interest rate received on the cash flows upon reinvestment

**Returns**

**out**

[float] Modified internal rate of return

### References

[1]

`numpy.nper`(*rate*, *pmt*, *pv*, *fv=0*, *when='end'*)

Compute the number of periodic payments.

Deprecated since version 1.18: `nper` is deprecated; for details, see NEP 32 [1]. Use the corresponding function in the numpy-financial library, https://pypi.org/project/numpy-financial.

`decimal.Decimal` type is not supported.

**Parameters**

**rate**

[array_like] Rate of interest (per period)

**pmt**

[array_like] Payment

**pv**

[array_like] Present value

**fv**

[array_like, optional] Future value

**when**

[{{'begin', 1}, {'end', 0}}, {string, int}, optional] When payments are due ('begin' (1) or 'end' (0))

### Notes

The number of periods `nper` is computed by solving the equation:

```
fv + pv*(1+rate)**nper + pmt*(1+rate*when)/rate*((1+rate)**nper-1) = 0
```

but if `rate = 0` then:

```
fv + pv + pmt*nper = 0
```

### References

[1]

### Examples

If you only had $150/month to pay towards the loan, how long would it take to pay-off a loan of $8,000 at 7% annual interest?

```
>>> print(np.round(np.nper(0.07/12, -150, 8000), 5))
64.07335
```

So, over 64 months would be required to pay off the loan.

The same analysis could be done with several different interest rates and/or payments and/or total amounts to produce an entire table.

```
>>> np.nper(*(np.ogrid[0.07/12: 0.08/12: 0.01/12,
...                    -150   : -99     : 50    ,
...                    8000   : 9001    : 1000]))
array([[[ 64.07334877,  74.06368256],
        [108.07548412, 127.99022654]],
       [[ 66.12443902,  76.87897353],
        [114.70165583, 137.90124779]]])
```

`numpy.`**`rate`**(*nper*, *pmt*, *pv*, *fv*, *when='end'*, *guess=None*, *tol=None*, *maxiter=100*)

Compute the rate of interest per period.

Deprecated since version 1.18: `rate` is deprecated; for details, see NEP 32 [1]. Use the corresponding function in the numpy-financial library, https://pypi.org/project/numpy-financial.

**Parameters**

**nper**

[array_like] Number of compounding periods

**pmt**

[array_like] Payment

**pv**

[array_like] Present value

**fv**

[array_like] Future value

**when**

[{{'begin', 1}, {'end', 0}}, {string, int}, optional] When payments are due ('begin' (1) or 'end' (0))

**guess**

[Number, optional] Starting guess for solving the rate of interest, default 0.1

**tol**

[Number, optional] Required tolerance for the solution, default 1e-6

**maxiter**

[int, optional] Maximum iterations in finding the solution

### Notes

The rate of interest is computed by iteratively solving the (non-linear) equation:

```
fv + pv*(1+rate)**nper + pmt*(1+rate*when)/rate * ((1+rate)**nper - 1) = 0
```

for `rate`.

### References

[1], [2]

## 4.13 Functional programming

| | |
|---|---|
| *apply_along_axis*(func1d, axis, arr, *args, …) | Apply a function to 1-D slices along the given axis. |
| *apply_over_axes*(func, a, axes) | Apply a function repeatedly over multiple axes. |
| *vectorize*(pyfunc[, otypes, doc, excluded, …]) | Generalized function class. |
| *frompyfunc*(func, nin, nout, *[, identity]) | Takes an arbitrary Python function and returns a NumPy ufunc. |
| *piecewise*(x, condlist, funclist, *args, **kw) | Evaluate a piecewise-defined function. |

`numpy.`**`apply_along_axis`**(*func1d*, *axis*, *arr*, **args*, ***kwargs*)

Apply a function to 1-D slices along the given axis.

Execute *func1d(a, *args, **kwargs)* where *func1d* operates on 1-D arrays and *a* is a 1-D slice of *arr* along *axis*.

This is equivalent to (but faster than) the following use of *ndindex* and *s_*, which sets each of `ii`, `jj`, and `kk` to a tuple of indices:

```
Ni, Nk = a.shape[:axis], a.shape[axis+1:]
for ii in ndindex(Ni):
    for kk in ndindex(Nk):
        f = func1d(arr[ii + s_[:,] + kk])
        Nj = f.shape
        for jj in ndindex(Nj):
            out[ii + jj + kk] = f[jj]
```

Equivalently, eliminating the inner loop, this can be expressed as:

```
Ni, Nk = a.shape[:axis], a.shape[axis+1:]
for ii in ndindex(Ni):
    for kk in ndindex(Nk):
        out[ii + s_[...,] + kk] = func1d(arr[ii + s_[:,] + kk])
```

**Parameters**

**func1d**

[function (M,) -> (Nj…)] This function should accept 1-D arrays. It is applied to 1-D slices of *arr* along the specified axis.

**axis**

[integer] Axis along which *arr* is sliced.

**arr**

[ndarray (Ni…, M, Nk…)] Input array.

**args**

[any] Additional arguments to *func1d*.

**kwargs**

[any] Additional named arguments to *func1d*.

New in version 1.9.0.

**Returns**

**out**

[ndarray (Ni…, Nj…, Nk…)] The output array. The shape of *out* is identical to the shape of *arr*, except along the *axis* dimension. This axis is removed, and replaced with new dimensions equal to the shape of the return value of *func1d*. So if *func1d* returns a scalar *out* will have one fewer dimensions than *arr*.

**See also:**

`apply_over_axes`

Apply a function repeatedly over multiple axes.

## Examples

```
>>> def my_func(a):
...     """Average first and last element of a 1-D array"""
...     return (a[0] + a[-1]) * 0.5
>>> b = np.array([[1,2,3], [4,5,6], [7,8,9]])
>>> np.apply_along_axis(my_func, 0, b)
array([4., 5., 6.])
>>> np.apply_along_axis(my_func, 1, b)
array([2.,  5.,  8.])
```

For a function that returns a 1D array, the number of dimensions in *outarr* is the same as *arr*.

```
>>> b = np.array([[8,1,7], [4,3,9], [5,2,6]])
>>> np.apply_along_axis(sorted, 1, b)
array([[1, 7, 8],
       [3, 4, 9],
       [2, 5, 6]])
```

For a function that returns a higher dimensional array, those dimensions are inserted in place of the *axis* dimension.

```
>>> b = np.array([[1,2,3], [4,5,6], [7,8,9]])
>>> np.apply_along_axis(np.diag, -1, b)
array([[[1, 0, 0],
        [0, 2, 0],
        [0, 0, 3]],
       [[4, 0, 0],
        [0, 5, 0],
        [0, 0, 6]],
```

(continues on next page)

(continued from previous page)

```
       [[7, 0, 0],
        [0, 8, 0],
        [0, 0, 9]]])
```

`numpy.`**`apply_over_axes`**(*func*, *a*, *axes*)

Apply a function repeatedly over multiple axes.

*func* is called as *res = func(a, axis)*, where *axis* is the first element of *axes*. The result *res* of the function call must have either the same dimensions as *a* or one less dimension. If *res* has one less dimension than *a*, a dimension is inserted before *axis*. The call to *func* is then repeated for each axis in *axes*, with *res* as the first argument.

**Parameters**

**func**

[function] This function must take two arguments, *func(a, axis)*.

**a**

[array_like] Input array.

**axes**

[array_like] Axes over which *func* is applied; the elements must be integers.

**Returns**

**apply_over_axis**

[ndarray] The output array. The number of dimensions is the same as *a*, but the shape can be different. This depends on whether *func* changes the shape of its output with respect to its input.

**See also:**

`apply_along_axis`

Apply a function to 1-D slices of an array along the given axis.

## Notes

This function is equivalent to tuple axis arguments to reorderable ufuncs with keepdims=True. Tuple axis arguments to ufuncs have been available since version 1.7.0.

## Examples

```
>>> a = np.arange(24).reshape(2,3,4)
>>> a
array([[[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [ 8,  9, 10, 11]],
       [[12, 13, 14, 15],
        [16, 17, 18, 19],
        [20, 21, 22, 23]]])
```

Sum over axes 0 and 2. The result has same number of dimensions as the original array:

```
>>> np.apply_over_axes(np.sum, a, [0,2])
array([[[ 60],
        [ 92],
        [124]]])
```

Tuple axis arguments to ufuncs are equivalent:

```
>>> np.sum(a, axis=(0,2), keepdims=True)
array([[[ 60],
        [ 92],
        [124]]])
```

**class** `numpy.`**`vectorize`**(*pyfunc*, *otypes=None*, *doc=None*, *excluded=None*, *cache=False*, *signature=None*)

Generalized function class.

Define a vectorized function which takes a nested sequence of objects or numpy arrays as inputs and returns a single numpy array or a tuple of numpy arrays. The vectorized function evaluates *pyfunc* over successive tuples of the input arrays like the python map function, except it uses the broadcasting rules of numpy.

The data type of the output of *vectorized* is determined by calling the function with the first element of the input. This can be avoided by specifying the *otypes* argument.

**Parameters**

**pyfunc**

[callable] A python function or method.

**otypes**

[str or list of dtypes, optional] The output data type. It must be specified as either a string of typecode characters or a list of data type specifiers. There should be one data type specifier for each output.

**doc**

[str, optional] The docstring for the function. If None, the docstring will be the `pyfunc.__doc__`.

**excluded**

[set, optional] Set of strings or integers representing the positional or keyword arguments for which the function will not be vectorized. These will be passed directly to *pyfunc* unmodified.

New in version 1.7.0.

**cache**

[bool, optional] If *True*, then cache the first function call that determines the number of outputs if *otypes* is not provided.

New in version 1.7.0.

**signature**

[string, optional] Generalized universal function signature, e.g., `(m,n),(n)->(m)` for vectorized matrix-vector multiplication. If provided, `pyfunc` will be called with (and expected to return) arrays with shapes given by the size of corresponding core dimensions. By default, `pyfunc` is assumed to take scalars as input and output.

New in version 1.12.0.

**Returns**

**vectorized**

[callable] Vectorized function.

**See also:**

*`frompyfunc`*

Takes an arbitrary Python function and returns a ufunc

## Notes

The `vectorize` function is provided primarily for convenience, not for performance. The implementation is essentially a for loop.

If *otypes* is not specified, then a call to the function with the first argument will be used to determine the number of outputs. The results of this call will be cached if *cache* is *True* to prevent calling the function twice. However, to implement the cache, the original function must be wrapped which will slow down subsequent calls, so only do this if your function is expensive.

The new keyword argument interface and *excluded* argument support further degrades performance.

## References

[R5cc1f1f25381-1]

## Examples

```
>>> def myfunc(a, b):
...     "Return a-b if a>b, otherwise return a+b"
...     if a > b:
...         return a - b
...     else:
...         return a + b
```

```
>>> vfunc = np.vectorize(myfunc)
>>> vfunc([1, 2, 3, 4], 2)
array([3, 4, 1, 2])
```

The docstring is taken from the input function to `vectorize` unless it is specified:

```
>>> vfunc.__doc__
'Return a-b if a>b, otherwise return a+b'
>>> vfunc = np.vectorize(myfunc, doc='Vectorized `myfunc`')
>>> vfunc.__doc__
'Vectorized `myfunc`'
```

The output type is determined by evaluating the first element of the input, unless it is specified:

```
>>> out = vfunc([1, 2, 3, 4], 2)
>>> type(out[0])
<class 'numpy.int64'>
>>> vfunc = np.vectorize(myfunc, otypes=[float])
>>> out = vfunc([1, 2, 3, 4], 2)
>>> type(out[0])
<class 'numpy.float64'>
```

The *excluded* argument can be used to prevent vectorizing over certain arguments. This can be useful for array-like arguments of a fixed length such as the coefficients for a polynomial as in `polyval`:

```
>>> def mypolyval(p, x):
...     _p = list(p)
...     res = _p.pop(0)
...     while _p:
...         res = res*x + _p.pop(0)
...     return res
>>> vpolyval = np.vectorize(mypolyval, excluded=['p'])
>>> vpolyval(p=[1, 2, 3], x=[0, 1])
array([3, 6])
```

Positional arguments may also be excluded by specifying their position:

```
>>> vpolyval.excluded.add(0)
>>> vpolyval([1, 2, 3], x=[0, 1])
array([3, 6])
```

The *signature* argument allows for vectorizing functions that act on non-scalar arrays of fixed length. For example, you can use it for a vectorized calculation of Pearson correlation coefficient and its p-value:

```
>>> import scipy.stats
>>> pearsonr = np.vectorize(scipy.stats.pearsonr,
...                 signature='(n),(n)->(),()')
>>> pearsonr([[0, 1, 2, 3]], [[1, 2, 3, 4], [4, 3, 2, 1]])
(array([ 1., -1.]), array([ 0.,  0.]))
```

Or for a vectorized convolution:

```
>>> convolve = np.vectorize(np.convolve, signature='(n),(m)->(k)')
>>> convolve(np.eye(4), [1, 2, 1])
array([[1., 2., 1., 0., 0., 0.],
       [0., 1., 2., 1., 0., 0.],
       [0., 0., 1., 2., 1., 0.],
       [0., 0., 0., 1., 2., 1.]])
```

## Methods

| | |
|---|---|
| `__call__`(self, *args, **kwargs) | Return arrays with the results of *pyfunc* broadcast (vectorized) over *args* and *kwargs* not in *excluded*. |

method

`vectorize.`**`__call__`**(*self*, **args*, ***kwargs*)
Return arrays with the results of *pyfunc* broadcast (vectorized) over *args* and *kwargs* not in *excluded*.

`numpy.`**`frompyfunc`**(*func*, *nin*, *nout*, *[, *identity* ])

Takes an arbitrary Python function and returns a NumPy ufunc.

Can be used, for example, to add broadcasting to a built-in Python function (see Examples section).

**Parameters**

**func**

[Python function object] An arbitrary Python function.

**nin**

[int] The number of input arguments.

**nout**

[int] The number of objects returned by *func*.

**identity**

[object, optional] The value to use for the *`identity`* attribute of the resulting object. If specified, this is equivalent to setting the underlying C `identity` field to `PyUFunc_IdentityValue`. If omitted, the identity is set to `PyUFunc_None`. Note that this is _not_ equivalent to setting the identity to `None`, which implies the operation is reorderable.

**Returns**

**out**

[ufunc] Returns a NumPy universal function (`ufunc`) object.

**See also:**

***`vectorize`***

Evaluates pyfunc over input arrays using broadcasting rules of numpy.

## Notes

The returned ufunc always returns PyObject arrays.

## Examples

Use frompyfunc to add broadcasting to the Python function `oct`:

```
>>> oct_array = np.frompyfunc(oct, 1, 1)
>>> oct_array(np.array((10, 30, 100)))
array(['0o12', '0o36', '0o144'], dtype=object)
>>> np.array((oct(10), oct(30), oct(100))) # for comparison
array(['0o12', '0o36', '0o144'], dtype='<U5')
```

`numpy.`**`piecewise`**(*x*, *condlist*, *funclist*, **args*, ***kw*)

Evaluate a piecewise-defined function.

Given a set of conditions and corresponding functions, evaluate each function on the input data wherever its condition is true.

**Parameters**

**x**

[ndarray or scalar] The input domain.

**condlist**

[list of bool arrays or bool scalars] Each boolean array corresponds to a function in *funclist*. Wherever *condlist[i]* is True, *funclist[i](x)* is used as the output value.

Each boolean array in *condlist* selects a piece of *x*, and should therefore be of the same shape as *x*.

The length of *condlist* must correspond to that of *funclist*. If one extra function is given, i.e. if `len(funclist) == len(condlist) + 1`, then that extra function is the default value, used wherever all conditions are false.

**funclist**

[list of callables, f(x,*args,**kw), or scalars] Each function is evaluated over *x* wherever its corresponding condition is True. It should take a 1d array as input and give an 1d array or a scalar value as output. If, instead of a callable, a scalar is provided then a constant function (`lambda x: scalar`) is assumed.

**args**

[tuple, optional] Any further arguments given to `piecewise` are passed to the functions upon execution, i.e., if called `piecewise(..., ..., 1, 'a')`, then each function is called as `f(x, 1, 'a')`.

**kw**

[dict, optional] Keyword arguments used in calling `piecewise` are passed to the functions upon execution, i.e., if called `piecewise(..., ..., alpha=1)`, then each function is called as `f(x, alpha=1)`.

**Returns**

**out**

[ndarray] The output is the same shape and type as x and is found by calling the functions in *funclist* on the appropriate portions of *x*, as defined by the boolean arrays in *condlist*. Portions not covered by any condition have a default value of 0.

**See also:**

`choose`, `select`, `where`

## Notes

This is similar to choose or select, except that functions are evaluated on elements of *x* that satisfy the corresponding condition from *condlist*.

The result is:

```
        |--
        |funclist[0](x[condlist[0]])
out =   |funclist[1](x[condlist[1]])
        |...
```

(continues on next page)

(continued from previous page)

```
        |funclist[n2](x[condlist[n2]])
        |--
```

### Examples

Define the sigma function, which is -1 for `x < 0` and +1 for `x >= 0`.

```
>>> x = np.linspace(-2.5, 2.5, 6)
>>> np.piecewise(x, [x < 0, x >= 0], [-1, 1])
array([-1., -1., -1.,  1.,  1.,  1.])
```

Define the absolute value, which is `-x` for `x <0` and `x` for `x >= 0`.

```
>>> np.piecewise(x, [x < 0, x >= 0], [lambda x: -x, lambda x: x])
array([2.5,  1.5,  0.5,  0.5,  1.5,  2.5])
```

Apply the same function to a scalar value.

```
>>> y = -2
>>> np.piecewise(y, [y < 0, y >= 0], [lambda x: -x, lambda x: x])
array(2)
```

# 4.14 NumPy-specific help functions

## 4.14.1 Finding help

| | |
|---|---|
| *`lookfor`*(what[, module, import_modules, …]) | Do a keyword search on docstrings. |

`numpy.`**`lookfor`**(*what*, *module=None*, *import_modules=True*, *regenerate=False*, *output=None*)

Do a keyword search on docstrings.

A list of objects that matched the search is displayed, sorted by relevance. All given keywords need to be found in the docstring for it to be returned as a result, but the order does not matter.

**Parameters**

**what**

[str] String containing words to look for.

**module**

[str or list, optional] Name of module(s) whose docstrings to go through.

**import_modules**

[bool, optional] Whether to import sub-modules in packages. Default is True.

**regenerate**

[bool, optional] Whether to re-generate the docstring cache. Default is False.

**output**

[file-like, optional] File-like object to write the output to. If omitted, use a pager.

**See also:**

*source*, *info*

### Notes

Relevance is determined only roughly, by checking if the keywords occur in the function name, at the start of a docstring, etc.

### Examples

```
>>> np.lookfor('binary representation')
Search results for 'binary representation'
------------------------------------------
numpy.binary_repr
    Return the binary representation of the input number as a string.
numpy.core.setup_common.long_double_representation
    Given a binary dump as given by GNU od -b, look for long double
numpy.base_repr
    Return a string representation of a number in the given base system.
...
```

## 4.14.2 Reading help

| | |
|---|---|
| *info*([object, maxwidth, output, toplevel]) | Get help information for a function, class, or module. |
| *source*(object[, output]) | Print or write to a file the source code for a NumPy object. |

`numpy.`**`info`**(*object=None, maxwidth=76, output=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='utf-8'>, toplevel='numpy'*)

Get help information for a function, class, or module.

**Parameters**

**object**

[object or str, optional] Input object or name to get information about. If `object` is a numpy object, its docstring is given. If it is a string, available modules are searched for matching objects. If None, information about *info* itself is returned.

**maxwidth**

[int, optional] Printing width.

**output**

[file like object, optional] File like object that the output is written to, default is `stdout`. The object has to be opened in 'w' or 'a' mode.

**toplevel**

[str, optional] Start search at this level.

**See also:**

*source*, *lookfor*

### Notes

When used interactively with an object, `np.info(obj)` is equivalent to `help(obj)` on the Python prompt or `obj?` on the IPython prompt.

### Examples

```
>>> np.info(np.polyval)
   polyval(p, x)
     Evaluate the polynomial p at x.
     ...
```

When using a string for `object` it is possible to get multiple results.

```
>>> np.info('fft')
     *** Found in numpy ***
Core FFT routines
...
     *** Found in numpy.fft ***
 fft(a, n=None, axis=-1)
...
     *** Repeat reference found in numpy.fft.fftpack ***
     *** Total of 3 references found. ***
```

`numpy.`**`source`**(*object*, *output=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='utf-8'>*)

Print or write to a file the source code for a NumPy object.

The source code is only returned for objects written in Python. Many functions and classes are defined in C and will therefore not return useful information.

**Parameters**

**object**

[numpy object] Input object. This can be any object (function, class, module, …).

**output**

[file object, optional] If *output* not supplied then source code is printed to screen (sys.stdout). File object must be created with either write 'w' or append 'a' modes.

**See also:**

*`lookfor`*, *`info`*

### Examples

```
>>> np.source(np.interp)
In file: /usr/lib/python2.6/dist-packages/numpy/lib/function_base.py
def interp(x, xp, fp, left=None, right=None):
    """.... (full docstring printed)"""
    if isinstance(x, (float, int, number)):
        return compiled_interp([x], xp, fp, left, right).item()
    else:
        return compiled_interp(x, xp, fp, left, right)
```

The source code is only returned for objects written in Python.

```
>>> np.source(np.array)
Not available for this object.
```

## 4.15 Indexing routines

**See also:**

*Indexing*

### 4.15.1 Generating index arrays

| | |
|---|---|
| `c_` | Translates slice objects to concatenation along the second axis. |
| `r_` | Translates slice objects to concatenation along the first axis. |
| `s_` | A nicer way to build up index tuples for arrays. |
| `nonzero`(a) | Return the indices of the elements that are non-zero. |
| `where`(condition, [x, y]) | Return elements chosen from *x* or *y* depending on *condition*. |
| `indices`(dimensions[, dtype, sparse]) | Return an array representing the indices of a grid. |
| `ix_`(*args) | Construct an open mesh from multiple sequences. |
| `ogrid` | *nd_grid* instance which returns an open multi-dimensional “meshgrid”. |
| `ravel_multi_index`(multi_index, dims[, mode, …]) | Converts a tuple of index arrays into an array of flat indices, applying boundary modes to the multi-index. |
| `unravel_index`(indices, shape[, order]) | Converts a flat index or array of flat indices into a tuple of coordinate arrays. |
| `diag_indices`(n[, ndim]) | Return the indices to access the main diagonal of an array. |
| `diag_indices_from`(arr) | Return the indices to access the main diagonal of an n-dimensional array. |
| `mask_indices`(n, mask_func[, k]) | Return the indices to access (n, n) arrays, given a masking function. |
| `tril_indices`(n[, k, m]) | Return the indices for the lower-triangle of an (n, m) array. |
| `tril_indices_from`(arr[, k]) | Return the indices for the lower-triangle of arr. |
| `triu_indices`(n[, k, m]) | Return the indices for the upper-triangle of an (n, m) array. |
| `triu_indices_from`(arr[, k]) | Return the indices for the upper-triangle of arr. |

`numpy.`**`c_ = <numpy.lib.index_tricks.CClass object>`**

Translates slice objects to concatenation along the second axis.

This is short-hand for `np.r_['-1,2,0', index expression]`, which is useful because of its common occurrence. In particular, arrays will be stacked along their last axis after being upgraded to at least 2-D with 1’s post-pended to the shape (column vectors made out of 1-D arrays).

**See also:**

***`column_stack`***

Stack 1-D arrays as columns into a 2-D array.

*r_*

For more detailed documentation.

### Examples

```
>>> np.c_[np.array([1,2,3]), np.array([4,5,6])]
array([[1, 4],
       [2, 5],
       [3, 6]])
>>> np.c_[np.array([[1,2,3]]), 0, 0, np.array([[4,5,6]])]
array([[1, 2, 3, ..., 4, 5, 6]])
```

`numpy.`**`r_ = <numpy.lib.index_tricks.RClass object>`**

Translates slice objects to concatenation along the first axis.

This is a simple way to build up arrays quickly. There are two use cases.

1. If the index expression contains comma separated arrays, then stack them along their first axis.
2. If the index expression contains slice notation or scalars then create a 1-D array with a range indicated by the slice notation.

If slice notation is used, the syntax `start:stop:step` is equivalent to `np.arange(start, stop, step)` inside of the brackets. However, if `step` is an imaginary number (i.e. 100j) then its integer portion is interpreted as a number-of-points desired and the start and stop are inclusive. In other words `start:stop:stepj` is interpreted as `np.linspace(start, stop, step, endpoint=1)` inside of the brackets. After expansion of slice notation, all comma separated sequences are concatenated together.

Optional character strings placed as the first element of the index expression can be used to change the output. The strings 'r' or 'c' result in matrix output. If the result is 1-D and 'r' is specified a 1 x N (row) matrix is produced. If the result is 1-D and 'c' is specified, then a N x 1 (column) matrix is produced. If the result is 2-D then both provide the same matrix result.

A string integer specifies which axis to stack multiple comma separated arrays along. A string of two comma-separated integers allows indication of the minimum number of dimensions to force each entry into as the second integer (the axis to concatenate along is still the first integer).

A string with three comma-separated integers allows specification of the axis to concatenate along, the minimum number of dimensions to force the entries to, and which axis should contain the start of the arrays which are less than the specified number of dimensions. In other words the third integer allows you to specify where the 1's should be placed in the shape of the arrays that have their shapes upgraded. By default, they are placed in the front of the shape tuple. The third argument allows you to specify where the start of the array should be instead. Thus, a third argument of '0' would place the 1's at the end of the array shape. Negative integers specify where in the new shape tuple the last dimension of upgraded arrays should be placed, so the default is '-1'.

**Parameters**

**Not a function, so takes no parameters**

**Returns**

**A concatenated ndarray or matrix.**

**See also:**

*concatenate*

Join a sequence of arrays along an existing axis.

*c_*

Translates slice objects to concatenation along the second axis.

### Examples

```
>>> np.r_[np.array([1,2,3]), 0, 0, np.array([4,5,6])]
array([1, 2, 3, ..., 4, 5, 6])
>>> np.r_[-1:1:6j, [0]*3, 5, 6]
array([-1. , -0.6, -0.2,  0.2,  0.6,  1. ,  0. ,  0. ,  0. ,  5. ,  6. ])
```

String integers specify the axis to concatenate along or the minimum number of dimensions to force entries into.

```
>>> a = np.array([[0, 1, 2], [3, 4, 5]])
>>> np.r_['-1', a, a] # concatenate along last axis
array([[0, 1, 2, 0, 1, 2],
       [3, 4, 5, 3, 4, 5]])
>>> np.r_['0,2', [1,2,3], [4,5,6]] # concatenate along first axis, dim>=2
array([[1, 2, 3],
       [4, 5, 6]])
```

```
>>> np.r_['0,2,0', [1,2,3], [4,5,6]]
array([[1],
       [2],
       [3],
       [4],
       [5],
       [6]])
>>> np.r_['1,2,0', [1,2,3], [4,5,6]]
array([[1, 4],
       [2, 5],
       [3, 6]])
```

Using 'r' or 'c' as a first string argument creates a matrix.

```
>>> np.r_['r',[1,2,3], [4,5,6]]
matrix([[1, 2, 3, 4, 5, 6]])
```

`numpy.`**`s_ = <numpy.lib.index_tricks.IndexExpression object>`**

A nicer way to build up index tuples for arrays.

**Note:** Use one of the two predefined instances `index_exp` or `s_` rather than directly using *IndexExpression*.

For any index combination, including slicing and axis insertion, `a[indices]` is the same as `a[np.index_exp[indices]]` for any array *a*. However, `np.index_exp[indices]` can be used anywhere in Python code and returns a tuple of slice objects that can be used in the construction of complex index expressions.

**Parameters**

**maketuple**

[bool] If True, always returns a tuple.

**See also:**

**`index_exp`**

Predefined instance that always returns a tuple: *index_exp = IndexExpression(maketuple=True)*.

**`s_`**

Predefined instance without tuple conversion: *s_ = IndexExpression(maketuple=False)*.

## Notes

You can do all this with *slice()* plus a few special objects, but there's a lot to remember and this version is simpler because it uses the standard array indexing syntax.

## Examples

```
>>> np.s_[2::2]
slice(2, None, 2)
>>> np.index_exp[2::2]
(slice(2, None, 2),)
```

```
>>> np.array([0, 1, 2, 3, 4])[np.s_[2::2]]
array([2, 4])
```

`numpy.`**`nonzero`**`(`*a*`)`

Return the indices of the elements that are non-zero.

Returns a tuple of arrays, one for each dimension of *a*, containing the indices of the non-zero elements in that dimension. The values in *a* are always tested and returned in row-major, C-style order.

To group the indices by element, rather than dimension, use *`argwhere`*, which returns a row for each non-zero element.

---

**Note:** When called on a zero-d array or scalar, `nonzero(a)` is treated as `nonzero(atleast1d(a))`.

Deprecated since version 1.17.0: Use *atleast1d* explicitly if this behavior is deliberate.

---

**Parameters**

**a**

[array_like] Input array.

**Returns**

**tuple_of_arrays**

[tuple] Indices of elements that are non-zero.

**See also:**

***`flatnonzero`***

Return indices that are non-zero in the flattened version of the input array.

***ndarray.nonzero***

Equivalent ndarray method.

***count_nonzero***

Counts the number of non-zero elements in the input array.

## Notes

While the nonzero values can be obtained with `a[nonzero(a)]`, it is recommended to use `x[x.astype(bool)]` or `x[x != 0]` instead, which will correctly handle 0-d arrays.

## Examples

```
>>> x = np.array([[3, 0, 0], [0, 4, 0], [5, 6, 0]])
>>> x
array([[3, 0, 0],
       [0, 4, 0],
       [5, 6, 0]])
>>> np.nonzero(x)
(array([0, 1, 2, 2]), array([0, 1, 0, 1]))
```

```
>>> x[np.nonzero(x)]
array([3, 4, 5, 6])
>>> np.transpose(np.nonzero(x))
array([[0, 0],
       [1, 1],
       [2, 0],
       [2, 1]])
```

A common use for `nonzero` is to find the indices of an array, where a condition is True. Given an array *a*, the condition *a* > 3 is a boolean array and since False is interpreted as 0, np.nonzero(a > 3) yields the indices of the *a* where the condition is true.

```
>>> a = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
>>> a > 3
array([[False, False, False],
       [ True,  True,  True],
       [ True,  True,  True]])
>>> np.nonzero(a > 3)
(array([1, 1, 1, 2, 2, 2]), array([0, 1, 2, 0, 1, 2]))
```

Using this result to index *a* is equivalent to using the mask directly:

```
>>> a[np.nonzero(a > 3)]
array([4, 5, 6, 7, 8, 9])
>>> a[a > 3]  # prefer this spelling
array([4, 5, 6, 7, 8, 9])
```

`nonzero` can also be called as a method of the array.

```
>>> (a > 3).nonzero()
(array([1, 1, 1, 2, 2, 2]), array([0, 1, 2, 0, 1, 2]))
```

`numpy.where(condition[, x, y])`

Return elements chosen from *x* or *y* depending on *condition*.

---

**Note:** When only *condition* is provided, this function is a shorthand for `np.asarray(condition).nonzero()`. Using *nonzero* directly should be preferred, as it behaves correctly for subclasses. The rest of this documentation covers only the case where all three arguments are provided.

---

**Parameters**

**condition**

[array_like, bool] Where True, yield *x*, otherwise yield *y*.

**x, y**

[array_like] Values from which to choose. *x*, *y* and *condition* need to be broadcastable to some shape.

**Returns**

**out**

[ndarray] An array with elements from *x* where *condition* is True, and elements from *y* elsewhere.

**See also:**

*choose*

***nonzero***

The function that is called when x and y are omitted

## Notes

If all the arrays are 1-D, *where* is equivalent to:

```
[xv if c else yv
 for c, xv, yv in zip(condition, x, y)]
```

## Examples

```
>>> a = np.arange(10)
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> np.where(a < 5, a, 10*a)
array([ 0,  1,  2,  3,  4, 50, 60, 70, 80, 90])
```

This can be used on multidimensional arrays too:

```
>>> np.where([[True, False], [True, True]],
...          [[1, 2], [3, 4]],
...          [[9, 8], [7, 6]])
array([[1, 8],
       [3, 4]])
```

The shapes of x, y, and the condition are broadcast together:

```
>>> x, y = np.ogrid[:3, :4]
>>> np.where(x < y, x, 10 + y)  # both x and 10+y are broadcast
array([[10,  0,  0,  0],
       [10, 11,  1,  1],
       [10, 11, 12,  2]])
```

```
>>> a = np.array([[0, 1, 2],
...               [0, 2, 4],
...               [0, 3, 6]])
>>> np.where(a < 4, a, -1)  # -1 is broadcast
array([[ 0,  1,  2],
       [ 0,  2, -1],
       [ 0,  3, -1]])
```

`numpy.`**`indices`**(*dimensions*, *dtype=<class 'int'>*, *sparse=False*)

Return an array representing the indices of a grid.

Compute an array where the subarrays contain index values 0, 1, … varying only along the corresponding axis.

**Parameters**

**dimensions**

[sequence of ints] The shape of the grid.

**dtype**

[dtype, optional] Data type of the result.

**sparse**

[boolean, optional] Return a sparse representation of the grid instead of a dense representation. Default is False.

New in version 1.17.

**Returns**

**grid**

[one ndarray or tuple of ndarrays]

**If sparse is False:**

Returns one array of grid indices, `grid.shape = (len(dimensions),) + tuple(dimensions)`.

**If sparse is True:**

Returns a tuple of arrays, with `grid[i].shape = (1, ..., 1, dimensions[i], 1, ..., 1)` with dimensions[i] in the ith place

**See also:**

*`mgrid`*, *`ogrid`*, *`meshgrid`*

## Notes

The output shape in the dense case is obtained by prepending the number of dimensions in front of the tuple of dimensions, i.e. if *dimensions* is a tuple `(r0, ..., rN-1)` of length `N`, the output shape is `(N, r0, ..., rN-1)`.

The subarrays `grid[k]` contains the N-D array of indices along the `k-th` axis. Explicitly:

```
grid[k, i0, i1, ..., iN-1] = ik
```

## Examples

```
>>> grid = np.indices((2, 3))
>>> grid.shape
(2, 2, 3)
>>> grid[0]        # row indices
array([[0, 0, 0],
       [1, 1, 1]])
>>> grid[1]        # column indices
array([[0, 1, 2],
       [0, 1, 2]])
```

The indices can be used as an index into an array.

```
>>> x = np.arange(20).reshape(5, 4)
>>> row, col = np.indices((2, 3))
>>> x[row, col]
array([[0, 1, 2],
       [4, 5, 6]])
```

Note that it would be more straightforward in the above example to extract the required elements directly with `x[:2, :3]`.

If sparse is set to true, the grid will be returned in a sparse representation.

```
>>> i, j = np.indices((2, 3), sparse=True)
>>> i.shape
(2, 1)
>>> j.shape
(1, 3)
>>> i        # row indices
array([[0],
       [1]])
>>> j        # column indices
array([[0, 1, 2]])
```

`numpy.`**`ix_`**`(`**args*`)`

Construct an open mesh from multiple sequences.

This function takes N 1-D sequences and returns N outputs with N dimensions each, such that the shape is 1 in all but one dimension and the dimension with the non-unit shape value cycles through all N dimensions.

Using *`ix_`* one can quickly construct index arrays that will index the cross product. `a[np.ix_([1,3],[2,5])]` returns the array `[[a[1,2] a[1,5]], [a[3,2] a[3,5]]]`.

**Parameters**

**args**

[1-D sequences] Each sequence should be of integer or boolean type. Boolean sequences will be interpreted as boolean masks for the corresponding dimension (equivalent to passing in `np.nonzero(boolean_sequence)`).

**Returns**

**out**

[tuple of ndarrays] N arrays with N dimensions each, with N the number of input sequences. Together these arrays form an open mesh.

**See also:**

*ogrid*, *mgrid*, *meshgrid*

## Examples

```
>>> a = np.arange(10).reshape(2, 5)
>>> a
array([[0, 1, 2, 3, 4],
       [5, 6, 7, 8, 9]])
>>> ixgrid = np.ix_([0, 1], [2, 4])
>>> ixgrid
(array([[0],
       [1]]), array([[2, 4]]))
>>> ixgrid[0].shape, ixgrid[1].shape
((2, 1), (1, 2))
>>> a[ixgrid]
array([[2, 4],
       [7, 9]])
```

```
>>> ixgrid = np.ix_([True, True], [2, 4])
>>> a[ixgrid]
array([[2, 4],
       [7, 9]])
>>> ixgrid = np.ix_([True, True], [False, False, True, False, True])
>>> a[ixgrid]
array([[2, 4],
       [7, 9]])
```

numpy.**ravel_multi_index**(*multi_index*, *dims*, *mode='raise'*, *order='C'*)

Converts a tuple of index arrays into an array of flat indices, applying boundary modes to the multi-index.

**Parameters**

**multi_index**

[tuple of array_like] A tuple of integer arrays, one array for each dimension.

**dims**

[tuple of ints] The shape of array into which the indices from `multi_index` apply.

**mode**

[{'raise', 'wrap', 'clip'}, optional] Specifies how out-of-bounds indices are handled. Can specify either one mode or a tuple of modes, one mode per index.

- 'raise' – raise an error (default)
- 'wrap' – wrap around
- 'clip' – clip to the range

In 'clip' mode, a negative index which would normally wrap will clip to 0 instead.

**order**

[{'C', 'F'}, optional] Determines whether the multi-index should be viewed as indexing in row-major (C-style) or column-major (Fortran-style) order.

**Returns**

**raveled_indices**

[ndarray] An array of indices into the flattened version of an array of dimensions `dims`.

**See also:**

*`unravel_index`*

## Notes

New in version 1.6.0.

## Examples

```
>>> arr = np.array([[3,6,6],[4,5,1]])
>>> np.ravel_multi_index(arr, (7,6))
array([22, 41, 37])
>>> np.ravel_multi_index(arr, (7,6), order='F')
array([31, 41, 13])
>>> np.ravel_multi_index(arr, (4,6), mode='clip')
array([22, 23, 19])
>>> np.ravel_multi_index(arr, (4,4), mode=('clip','wrap'))
array([12, 13, 13])
```

```
>>> np.ravel_multi_index((3,1,4,1), (6,7,8,9))
1621
```

`numpy.`**`unravel_index`**(*indices*, *shape*, *order='C'*)

Converts a flat index or array of flat indices into a tuple of coordinate arrays.

**Parameters**

**indices**

[array_like] An integer array whose elements are indices into the flattened version of an array of dimensions `shape`. Before version 1.6.0, this function accepted just one index value.

**shape**

[tuple of ints] The shape of the array to use for unraveling `indices`.

Changed in version 1.16.0: Renamed from `dims` to `shape`.

**order**

[{'C', 'F'}, optional] Determines whether the indices should be viewed as indexing in row-major (C-style) or column-major (Fortran-style) order.

New in version 1.6.0.

**Returns**

**unraveled_coords**

[tuple of ndarray] Each array in the tuple has the same shape as the `indices` array.

**See also:**

*`ravel_multi_index`*

## Examples

```
>>> np.unravel_index([22, 41, 37], (7,6))
(array([3, 6, 6]), array([4, 5, 1]))
>>> np.unravel_index([31, 41, 13], (7,6), order='F')
(array([3, 6, 6]), array([4, 5, 1]))
```

```
>>> np.unravel_index(1621, (6,7,8,9))
(3, 1, 4, 1)
```

`numpy.`**`diag_indices`**`(`*n*, *ndim=2*`)`

Return the indices to access the main diagonal of an array.

This returns a tuple of indices that can be used to access the main diagonal of an array *a* with `a.ndim >= 2` dimensions and shape (n, n, …, n). For `a.ndim = 2` this is the usual diagonal, for `a.ndim > 2` this is the set of indices to access `a[i, i, ..., i]` for `i = [0..n-1]`.

**Parameters**

**n**

[int] The size, along each dimension, of the arrays for which the returned indices can be used.

**ndim**

[int, optional] The number of dimensions.

**See also:**

*`diag_indices_from`*

### Notes

New in version 1.4.0.

### Examples

Create a set of indices to access the diagonal of a (4, 4) array:

```
>>> di = np.diag_indices(4)
>>> di
(array([0, 1, 2, 3]), array([0, 1, 2, 3]))
>>> a = np.arange(16).reshape(4, 4)
>>> a
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15]])
>>> a[di] = 100
>>> a
array([[100,   1,   2,   3],
       [  4, 100,   6,   7],
       [  8,   9, 100,  11],
       [ 12,  13,  14, 100]])
```

Now, we create indices to manipulate a 3-D array:

```
>>> d3 = np.diag_indices(2, 3)
>>> d3
(array([0, 1]), array([0, 1]), array([0, 1]))
```

And use it to set the diagonal of an array of zeros to 1:

```
>>> a = np.zeros((2, 2, 2), dtype=int)
>>> a[d3] = 1
>>> a
array([[[1, 0],
        [0, 0]],
       [[0, 0],
        [0, 1]]])
```

`numpy.`**`diag_indices_from`**(*arr*)

Return the indices to access the main diagonal of an n-dimensional array.

See *`diag_indices`* for full details.

**Parameters**

**arr**

[array, at least 2-D]

**See also:**

*`diag_indices`*

### Notes

New in version 1.4.0.

`numpy.`**`mask_indices`**`(`*n*, *mask_func*, *k=0*`)`

Return the indices to access (n, n) arrays, given a masking function.

Assume *mask_func* is a function that, for a square array a of size `(n, n)` with a possible offset argument *k*, when called as `mask_func(a, k)` returns a new array with zeros in certain locations (functions like `triu` or `tril` do precisely this). Then this function returns the indices where the non-zero values would be located.

**Parameters**

**n**

[int] The returned indices will be valid to access arrays of shape (n, n).

**mask_func**

[callable] A function whose call signature is similar to that of `triu`, `tril`. That is, `mask_func(x, k)` returns a boolean array, shaped like *x*. *k* is an optional argument to the function.

**k**

[scalar] An optional argument which is passed through to *mask_func*. Functions like `triu`, `tril` take a second argument that is interpreted as an offset.

**Returns**

**indices**

[tuple of arrays.] The *n* arrays of indices corresponding to the locations where `mask_func(np.ones((n, n)), k)` is True.

**See also:**

`triu`, `tril`, `triu_indices`, `tril_indices`

### Notes

New in version 1.4.0.

### Examples

These are the indices that would allow you to access the upper triangular part of any 3x3 array:

```
>>> iu = np.mask_indices(3, np.triu)
```

For example, if *a* is a 3x3 array:

```
>>> a = np.arange(9).reshape(3, 3)
>>> a
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
>>> a[iu]
array([0, 1, 2, 4, 5, 8])
```

An offset can be passed also to the masking function. This gets us the indices starting on the first diagonal right of the main one:

```
>>> iu1 = np.mask_indices(3, np.triu, 1)
```

with which we now extract only three elements:

```
>>> a[iu1]
array([1, 2, 5])
```

`numpy.`**`tril_indices`**`(`*n*, *k=0*, *m=None*`)`

Return the indices for the lower-triangle of an (n, m) array.

**Parameters**

**n**

[int] The row dimension of the arrays for which the returned indices will be valid.

**k**

[int, optional] Diagonal offset (see `tril` for details).

**m**

[int, optional] New in version 1.9.0.

The column dimension of the arrays for which the returned arrays will be valid. By default *m* is taken equal to *n*.

**Returns**

**inds**

[tuple of arrays] The indices for the triangle. The returned tuple contains two arrays, each with the indices along one dimension of the array.

**See also:**

**`triu_indices`**

similar function, for upper-triangular.

**`mask_indices`**

generic function accepting an arbitrary mask function.

`tril`, `triu`

## Notes

New in version 1.4.0.

### Examples

Compute two different sets of indices to access 4x4 arrays, one for the lower triangular part starting at the main diagonal, and one starting two diagonals further right:

```
>>> il1 = np.tril_indices(4)
>>> il2 = np.tril_indices(4, 2)
```

Here is how they can be used with a sample array:

```
>>> a = np.arange(16).reshape(4, 4)
>>> a
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15]])
```

Both for indexing:

```
>>> a[il1]
array([ 0,  4,  5, ..., 13, 14, 15])
```

And for assigning values:

```
>>> a[il1] = -1
>>> a
array([[-1,  1,  2,  3],
       [-1, -1,  6,  7],
       [-1, -1, -1, 11],
       [-1, -1, -1, -1]])
```

These cover almost the whole array (two diagonals right of the main one):

```
>>> a[il2] = -10
>>> a
array([[-10, -10, -10,   3],
       [-10, -10, -10, -10],
       [-10, -10, -10, -10],
       [-10, -10, -10, -10]])
```

`numpy.`**`tril_indices_from`**(*arr*, *k=0*)

Return the indices for the lower-triangle of arr.

See `tril_indices` for full details.

**Parameters**

**arr**

[array_like] The indices will be valid for square arrays whose dimensions are the same as arr.

**k**

[int, optional] Diagonal offset (see `tril` for details).

**See also:**

`tril_indices`, `tril`

## Notes

New in version 1.4.0.

`numpy.`**`triu_indices`**(*n*, *k=0*, *m=None*)

Return the indices for the upper-triangle of an (n, m) array.

**Parameters**

**n**

[int] The size of the arrays for which the returned indices will be valid.

**k**

[int, optional] Diagonal offset (see `triu` for details).

**m**

[int, optional] New in version 1.9.0.

The column dimension of the arrays for which the returned arrays will be valid. By default *m* is taken equal to *n*.

**Returns**

**inds**

[tuple, shape(2) of ndarrays, shape(*n*)] The indices for the triangle. The returned tuple contains two arrays, each with the indices along one dimension of the array. Can be used to slice a ndarray of shape(*n*, *n*).

**See also:**

***`tril_indices`***

similar function, for lower-triangular.

***`mask_indices`***

generic function accepting an arbitrary mask function.

`triu`, `tril`

## Notes

New in version 1.4.0.

## Examples

Compute two different sets of indices to access 4x4 arrays, one for the upper triangular part starting at the main diagonal, and one starting two diagonals further right:

```
>>> iu1 = np.triu_indices(4)
>>> iu2 = np.triu_indices(4, 2)
```

Here is how they can be used with a sample array:

```
>>> a = np.arange(16).reshape(4, 4)
>>> a
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15]])
```

Both for indexing:

```
>>> a[iu1]
array([ 0,  1,  2, ..., 10, 11, 15])
```

And for assigning values:

```
>>> a[iu1] = -1
>>> a
array([[-1, -1, -1, -1],
       [ 4, -1, -1, -1],
       [ 8,  9, -1, -1],
       [12, 13, 14, -1]])
```

These cover only a small part of the whole array (two diagonals right of the main one):

```
>>> a[iu2] = -10
>>> a
array([[ -1,  -1, -10, -10],
       [  4,  -1,  -1, -10],
       [  8,   9,  -1,  -1],
       [ 12,  13,  14,  -1]])
```

`numpy.`**`triu_indices_from`**`(`*arr*, *k=0*`)`

Return the indices for the upper-triangle of arr.

See *`triu_indices`* for full details.

**Parameters**

**arr**

[ndarray, shape(N, N)] The indices will be valid for square arrays.

**k**

[int, optional] Diagonal offset (see *`triu`* for details).

**Returns**

**triu_indices_from**

[tuple, shape(2) of ndarray, shape(N)] Indices for the upper-triangle of *arr*.

**See also:**

*`triu_indices`*, *`triu`*

### Notes

New in version 1.4.0.

## 4.15.2 Indexing-like operations

| | |
|---|---|
| *take*(a, indices[, axis, out, mode]) | Take elements from an array along an axis. |
| *take_along_axis*(arr, indices, axis) | Take values from the input array by matching 1d index and data slices. |
| *choose*(a, choices[, out, mode]) | Construct an array from an index array and a set of arrays to choose from. |
| *compress*(condition, a[, axis, out]) | Return selected slices of an array along given axis. |
| *diag*(v[, k]) | Extract a diagonal or construct a diagonal array. |
| *diagonal*(a[, offset, axis1, axis2]) | Return specified diagonals. |
| *select*(condlist, choicelist[, default]) | Return an array drawn from elements in choicelist, depending on conditions. |
| *lib.stride_tricks.as_strided*(x[, shape, …]) | Create a view into the array with the given shape and strides. |

`numpy.`**`take`**(*a*, *indices*, *axis=None*, *out=None*, *mode='raise'*)

Take elements from an array along an axis.

When axis is not None, this function does the same thing as "fancy" indexing (indexing arrays using arrays); however, it can be easier to use if you need elements along a given axis. A call such as `np.take(arr, indices, axis=3)` is equivalent to `arr[:,:,:,indices,...]`.

Explained without fancy indexing, this is equivalent to the following use of *ndindex*, which sets each of `ii`, `jj`, and `kk` to a tuple of indices:

```
Ni, Nk = a.shape[:axis], a.shape[axis+1:]
Nj = indices.shape
for ii in ndindex(Ni):
    for jj in ndindex(Nj):
        for kk in ndindex(Nk):
            out[ii + jj + kk] = a[ii + (indices[jj],) + kk]
```

**Parameters**

**a**

[array_like (Ni…, M, Nk…)] The source array.

**indices**

[array_like (Nj…)] The indices of the values to extract.

New in version 1.8.0.

Also allow scalars for indices.

**axis**

[int, optional] The axis over which to select values. By default, the flattened input array is used.

**out**

[ndarray, optional (Ni…, Nj…, Nk…)] If provided, the result will be placed in this array. It should be of the appropriate shape and dtype. Note that *out* is always buffered if *mode='raise'*; use other modes for better performance.

**mode**

[{'raise', 'wrap', 'clip'}, optional] Specifies how out-of-bounds indices will behave.

- 'raise' – raise an error (default)
- 'wrap' – wrap around
- 'clip' – clip to the range

'clip' mode means that all indices that are too large are replaced by the index that addresses the last element along that axis. Note that this disables indexing with negative numbers.

**Returns**

**out**

[ndarray (Ni…, Nj…, Nk…)] The returned array has the same type as *a*.

**See also:**

**`compress`**

Take elements using a boolean mask

**`ndarray.take`**

equivalent method

**`take_along_axis`**

Take elements by matching the array and the index arrays

## Notes

By eliminating the inner loop in the description above, and using `s_` to build simple slice objects, `take` can be expressed in terms of applying fancy indexing to each 1-d slice:

```
Ni, Nk = a.shape[:axis], a.shape[axis+1:]
for ii in ndindex(Ni):
    for kk in ndindex(Nj):
        out[ii + s_[...,] + kk] = a[ii + s_[:,] + kk][indices]
```

For this reason, it is equivalent to (but faster than) the following use of `apply_along_axis`:

```
out = np.apply_along_axis(lambda a_1d: a_1d[indices], axis, a)
```

### Examples

```
>>> a = [4, 3, 5, 7, 6, 8]
>>> indices = [0, 1, 4]
>>> np.take(a, indices)
array([4, 3, 6])
```

In this example if *a* is an ndarray, “fancy” indexing can be used.

```
>>> a = np.array(a)
>>> a[indices]
array([4, 3, 6])
```

If *`indices`* is not one dimensional, the output also has these dimensions.

```
>>> np.take(a, [[0, 1], [2, 3]])
array([[4, 3],
       [5, 7]])
```

`numpy.`**`take_along_axis`**(*arr*, *indices*, *axis*)

Take values from the input array by matching 1d index and data slices.

This iterates over matching 1d slices oriented along the specified axis in the index and data arrays, and uses the former to look up values in the latter. These slices can be different lengths.

Functions returning an index along an axis, like *`argsort`* and *`argpartition`*, produce suitable indices for this function.

New in version 1.15.0.

**Parameters**

**arr: ndarray (Ni…, M, Nk…)**

Source array

**indices: ndarray (Ni…, J, Nk…)**

Indices to take along each 1d slice of *arr*. This must match the dimension of arr, but dimensions Ni and Nj only need to broadcast against *arr*.

**axis: int**

The axis to take 1d slices along. If axis is None, the input array is treated as if it had first been flattened to 1d, for consistency with *`sort`* and *`argsort`*.

**Returns**

**out: ndarray (Ni…, J, Nk…)**

The indexed result.

**See also:**

***`take`***

Take along an axis, using the same indices for every 1d slice

***`put_along_axis`***

Put values into the destination array by matching 1d index and data slices

## Notes

This is equivalent to (but faster than) the following use of *ndindex* and *s_*, which sets each of `ii` and `kk` to a tuple of indices:

```
Ni, M, Nk = a.shape[:axis], a.shape[axis], a.shape[axis+1:]
J = indices.shape[axis]  # Need not equal M
out = np.empty(Ni + (J,) + Nk)

for ii in ndindex(Ni):
    for kk in ndindex(Nk):
        a_1d       = a      [ii + s_[:,] + kk]
        indices_1d = indices[ii + s_[:,] + kk]
        out_1d     = out    [ii + s_[:,] + kk]
        for j in range(J):
            out_1d[j] = a_1d[indices_1d[j]]
```

Equivalently, eliminating the inner loop, the last two lines would be:

```
out_1d[:] = a_1d[indices_1d]
```

## Examples

For this sample array

```
>>> a = np.array([[10, 30, 20], [60, 40, 50]])
```

We can sort either by using sort directly, or argsort and this function

```
>>> np.sort(a, axis=1)
array([[10, 20, 30],
       [40, 50, 60]])
>>> ai = np.argsort(a, axis=1); ai
array([[0, 2, 1],
       [1, 2, 0]])
>>> np.take_along_axis(a, ai, axis=1)
array([[10, 20, 30],
       [40, 50, 60]])
```

The same works for max and min, if you expand the dimensions:

```
>>> np.expand_dims(np.max(a, axis=1), axis=1)
array([[30],
       [60]])
>>> ai = np.expand_dims(np.argmax(a, axis=1), axis=1)
>>> ai
array([[1],
       [0]])
>>> np.take_along_axis(a, ai, axis=1)
array([[30],
       [60]])
```

If we want to get the max and min at the same time, we can stack the indices first

```
>>> ai_min = np.expand_dims(np.argmin(a, axis=1), axis=1)
>>> ai_max = np.expand_dims(np.argmax(a, axis=1), axis=1)
```

(continues on next page)

(continued from previous page)

```
>>> ai = np.concatenate([ai_min, ai_max], axis=1)
>>> ai
array([[0, 1],
       [1, 0]])
>>> np.take_along_axis(a, ai, axis=1)
array([[10, 30],
       [40, 60]])
```

`numpy.`**`choose`**(*a*, *choices*, *out=None*, *mode='raise'*)

Construct an array from an index array and a set of arrays to choose from.

First of all, if confused or uncertain, definitely look at the Examples - in its full generality, this function is less simple than it might seem from the following code description (below ndi = `numpy.lib.index_tricks`):

`np.choose(a,c) == np.array([c[a[I]][I] for I in ndi.ndindex(a.shape)]).`

But this omits some subtleties. Here is a fully general summary:

Given an "index" array (*a*) of integers and a sequence of *n* arrays (*choices*), *a* and each choice array are first broadcast, as necessary, to arrays of a common shape; calling these *Ba* and *Bchoices[i], i = 0,…,n-1* we have that, necessarily, `Ba.shape == Bchoices[i].shape` for each *i*. Then, a new array with shape `Ba.shape` is created as follows:

- if `mode=raise` (the default), then, first of all, each element of *a* (and thus *Ba*) must be in the range *[0, n-1]*; now, suppose that *i* (in that range) is the value at the *(j0, j1, ..., jm)* position in *Ba* - then the value at the same position in the new array is the value in *Bchoices[i]* at that same position;
- if `mode=wrap`, values in *a* (and thus *Ba*) may be any (signed) integer; modular arithmetic is used to map integers outside the range *[0, n-1]* back into that range; and then the new array is constructed as above;
- if `mode=clip`, values in *a* (and thus *Ba*) may be any (signed) integer; negative integers are mapped to 0; values greater than *n-1* are mapped to *n-1*; and then the new array is constructed as above.

**Parameters**

**a**

[int array] This array must contain integers in *[0, n-1]*, where *n* is the number of choices, unless `mode=wrap` or `mode=clip`, in which cases any integers are permissible.

**choices**

[sequence of arrays] Choice arrays. *a* and all of the choices must be broadcastable to the same shape. If *choices* is itself an array (not recommended), then its outermost dimension (i.e., the one corresponding to `choices.shape[0]`) is taken as defining the "sequence".

**out**

[array, optional] If provided, the result will be inserted into this array. It should be of the appropriate shape and dtype. Note that *out* is always buffered if *mode='raise'*; use other modes for better performance.

**mode**

[{'raise' (default), 'wrap', 'clip'}, optional] Specifies how indices outside *[0, n-1]* will be treated:

- 'raise' : an exception is raised
- 'wrap' : value becomes value mod *n*
- 'clip' : values < 0 are mapped to 0, values > n-1 are mapped to n-1

**Returns**

**merged_array**

[array] The merged result.

**Raises**

**ValueError: shape mismatch**

If *a* and each choice array are not all broadcastable to the same shape.

**See also:**

**`ndarray.choose`**

equivalent method

**`numpy.take_along_axis`**

Preferable if *choices* is an array

## Notes

To reduce the chance of misinterpretation, even though the following "abuse" is nominally supported, *choices* should neither be, nor be thought of as, a single array, i.e., the outermost sequence-like container should be either a list or a tuple.

## Examples

```
>>> choices = [[0, 1, 2, 3], [10, 11, 12, 13],
...   [20, 21, 22, 23], [30, 31, 32, 33]]
>>> np.choose([2, 3, 1, 0], choices
... # the first element of the result will be the first element of the
... # third (2+1) "array" in choices, namely, 20; the second element
... # will be the second element of the fourth (3+1) choice array, i.e.,
... # 31, etc.
... )
array([20, 31, 12,  3])
>>> np.choose([2, 4, 1, 0], choices, mode='clip') # 4 goes to 3 (4-1)
array([20, 31, 12,  3])
>>> # because there are 4 choice arrays
>>> np.choose([2, 4, 1, 0], choices, mode='wrap') # 4 goes to (4 mod 4)
array([20,  1, 12,  3])
>>> # i.e., 0
```

A couple examples illustrating how choose broadcasts:

```
>>> a = [[1, 0, 1], [0, 1, 0], [1, 0, 1]]
>>> choices = [-10, 10]
>>> np.choose(a, choices)
array([[ 10, -10,  10],
       [-10,  10, -10],
       [ 10, -10,  10]])
```

```
>>> # With thanks to Anne Archibald
>>> a = np.array([0, 1]).reshape((2,1,1))
>>> c1 = np.array([1, 2, 3]).reshape((1,3,1))
>>> c2 = np.array([-1, -2, -3, -4, -5]).reshape((1,1,5))
>>> np.choose(a, (c1, c2)) # result is 2x3x5, res[0,:,:]=c1, res[1,:,:]=c2
array([[[ 1,  1,  1,  1,  1],
        [ 2,  2,  2,  2,  2],
        [ 3,  3,  3,  3,  3]],
       [[-1, -2, -3, -4, -5],
        [-1, -2, -3, -4, -5],
        [-1, -2, -3, -4, -5]]])
```

`numpy.`**`compress`**(*condition*, *a*, *axis=None*, *out=None*)

Return selected slices of an array along given axis.

When working along a given axis, a slice along that axis is returned in *output* for each index where *condition* evaluates to True. When working on a 1-D array, `compress` is equivalent to `extract`.

**Parameters**

**condition**

[1-D array of bools] Array that selects which entries to return. If len(condition) is less than the size of *a* along the given axis, then output is truncated to the length of the condition array.

**a**

[array_like] Array from which to extract a part.

**axis**

[int, optional] Axis along which to take slices. If None (default), work on the flattened array.

**out**

[ndarray, optional] Output array. Its type is preserved and it must be of the right shape to hold the output.

**Returns**

**compressed_array**

[ndarray] A copy of *a* without the slices along axis for which *condition* is false.

**See also:**

`take`, `choose`, `diag`, `diagonal`, `select`

**`ndarray.compress`**

Equivalent method in ndarray

**`np.extract`**

Equivalent method when working on 1-D arrays

`ufuncs-output-type`

### Examples

```
>>> a = np.array([[1, 2], [3, 4], [5, 6]])
>>> a
array([[1, 2],
       [3, 4],
       [5, 6]])
>>> np.compress([0, 1], a, axis=0)
array([[3, 4]])
>>> np.compress([False, True, True], a, axis=0)
array([[3, 4],
       [5, 6]])
>>> np.compress([False, True], a, axis=1)
array([[2],
       [4],
       [6]])
```

Working on the flattened array does not return slices along an axis but selects elements.

```
>>> np.compress([False, True], a)
array([2])
```

`numpy.`**`diagonal`**(*a*, *offset=0*, *axis1=0*, *axis2=1*)

Return specified diagonals.

If *a* is 2-D, returns the diagonal of *a* with the given offset, i.e., the collection of elements of the form `a[i, i+offset]`. If *a* has more than two dimensions, then the axes specified by *axis1* and *axis2* are used to determine the 2-D sub-array whose diagonal is returned. The shape of the resulting array can be determined by removing *axis1* and *axis2* and appending an index to the right equal to the size of the resulting diagonals.

In versions of NumPy prior to 1.7, this function always returned a new, independent array containing a copy of the values in the diagonal.

In NumPy 1.7 and 1.8, it continues to return a copy of the diagonal, but depending on this fact is deprecated. Writing to the resulting array continues to work as it used to, but a FutureWarning is issued.

Starting in NumPy 1.9 it returns a read-only view on the original array. Attempting to write to the resulting array will produce an error.

In some future release, it will return a read/write view and writing to the returned array will alter your original array. The returned array will have the same type as the input array.

If you don't write to the array returned by this function, then you can just ignore all of the above.

If you depend on the current behavior, then we suggest copying the returned array explicitly, i.e., use `np.diagonal(a).copy()` instead of just `np.diagonal(a)`. This will work with both past and future versions of NumPy.

**Parameters**

**a**

[array_like] Array from which the diagonals are taken.

**offset**

[int, optional] Offset of the diagonal from the main diagonal. Can be positive or negative. Defaults to main diagonal (0).

**axis1**

[int, optional] Axis to be used as the first axis of the 2-D sub-arrays from which the diagonals should be taken. Defaults to first axis (0).

**axis2**

[int, optional] Axis to be used as the second axis of the 2-D sub-arrays from which the diagonals should be taken. Defaults to second axis (1).

**Returns**

**array_of_diagonals**

[ndarray] If *a* is 2-D, then a 1-D array containing the diagonal and of the same type as *a* is returned unless *a* is a `matrix`, in which case a 1-D array rather than a (2-D) `matrix` is returned in order to maintain backward compatibility.

If `a.ndim > 2`, then the dimensions specified by *axis1* and *axis2* are removed, and a new axis inserted at the end corresponding to the diagonal.

**Raises**

**ValueError**

If the dimension of *a* is less than 2.

**See also:**

**`diag`**

MATLAB work-a-like for 1-D and 2-D arrays.

**`diagflat`**

Create diagonal arrays.

**`trace`**

Sum along diagonals.

## Examples

```
>>> a = np.arange(4).reshape(2,2)
>>> a
array([[0, 1],
       [2, 3]])
>>> a.diagonal()
array([0, 3])
>>> a.diagonal(1)
array([1])
```

A 3-D example:

```
>>> a = np.arange(8).reshape(2,2,2); a
array([[[0, 1],
        [2, 3]],
       [[4, 5],
        [6, 7]]])
>>> a.diagonal(0,  # Main diagonals of two arrays created by skipping
```

(continues on next page)

(continued from previous page)

```
...             0,  # across the outer(left)-most axis last and
...             1)  # the "middle" (row) axis first.
array([[0, 6],
       [1, 7]])
```

The sub-arrays whose main diagonals we just obtained; note that each corresponds to fixing the right-most (column) axis, and that the diagonals are "packed" in rows.

```
>>> a[:,:,0]  # main diagonal is [0 6]
array([[0, 2],
       [4, 6]])
>>> a[:,:,1]  # main diagonal is [1 7]
array([[1, 3],
       [5, 7]])
```

The anti-diagonal can be obtained by reversing the order of elements using either `numpy.flipud` or `numpy.fliplr`.

```
>>> a = np.arange(9).reshape(3, 3)
>>> a
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
>>> np.fliplr(a).diagonal()  # Horizontal flip
array([2, 4, 6])
>>> np.flipud(a).diagonal()  # Vertical flip
array([6, 4, 2])
```

Note that the order in which the diagonal is retrieved varies depending on the flip function.

`numpy.`**`select`**(*condlist*, *choicelist*, *default=0*)

Return an array drawn from elements in choicelist, depending on conditions.

**Parameters**

**condlist**

[list of bool ndarrays] The list of conditions which determine from which array in *choicelist* the output elements are taken. When multiple conditions are satisfied, the first one encountered in *condlist* is used.

**choicelist**

[list of ndarrays] The list of arrays from which the output elements are taken. It has to be of the same length as *condlist*.

**default**

[scalar, optional] The element inserted in *output* when all conditions evaluate to False.

**Returns**

**output**

[ndarray] The output at position m is the m-th element of the array in *choicelist* where the m-th element of the corresponding array in *condlist* is True.

**See also:**

**`where`**

Return elements from one of two arrays depending on condition.

`take`, `choose`, `compress`, `diag`, `diagonal`

### Examples

```
>>> x = np.arange(10)
>>> condlist = [x<3, x>5]
>>> choicelist = [x, x**2]
>>> np.select(condlist, choicelist)
array([ 0,  1,  2, ..., 49, 64, 81])
```

`numpy.lib.stride_tricks.`**`as_strided`**`(`*x*, *shape=None*, *strides=None*, *subok=False*, *writeable=True*`)`

Create a view into the array with the given shape and strides.

**Warning:** This function has to be used with extreme care, see notes.

**Parameters**

**x**

[ndarray] Array to create a new.

**shape**

[sequence of int, optional] The shape of the new array. Defaults to `x.shape`.

**strides**

[sequence of int, optional] The strides of the new array. Defaults to `x.strides`.

**subok**

[bool, optional] New in version 1.10.

If True, subclasses are preserved.

**writeable**

[bool, optional] New in version 1.12.

If set to False, the returned array will always be readonly. Otherwise it will be writable if the original array was. It is advisable to set this to False if possible (see Notes).

**Returns**

**view**

[ndarray]

**See also:**

**`broadcast_to`**

broadcast an array to a given shape.

**`reshape`**

reshape an array.

### Notes

`as_strided` creates a view into the array given the exact strides and shape. This means it manipulates the internal data structure of ndarray and, if done incorrectly, the array elements can point to invalid memory and can corrupt results or crash your program. It is advisable to always use the original `x.strides` when calculating new strides to avoid reliance on a contiguous memory layout.

Furthermore, arrays created with this function often contain self overlapping memory, so that two elements are identical. Vectorized write operations on such arrays will typically be unpredictable. They may even give different results for small, large, or transposed arrays. Since writing to these arrays has to be tested and done with great care, you may want to use `writeable=False` to avoid accidental write operations.

For these reasons it is advisable to avoid `as_strided` when possible.

## 4.15.3 Inserting data into arrays

| | |
|---|---|
| `place`(arr, mask, vals) | Change elements of an array based on conditional and input values. |
| `put`(a, ind, v[, mode]) | Replaces specified elements of an array with given values. |
| `put_along_axis`(arr, indices, values, axis) | Put values into the destination array by matching 1d index and data slices. |
| `putmask`(a, mask, values) | Changes elements of an array based on conditional and input values. |
| `fill_diagonal`(a, val[, wrap]) | Fill the main diagonal of the given array of any dimensionality. |

`numpy.`**`place`**(*arr*, *mask*, *vals*)

Change elements of an array based on conditional and input values.

Similar to `np.copyto(arr, vals, where=mask)`, the difference is that `place` uses the first N elements of *vals*, where N is the number of True values in *mask*, while `copyto` uses the elements where *mask* is True.

Note that `extract` does the exact opposite of `place`.

**Parameters**

**arr**

[ndarray] Array to put data into.

**mask**

[array_like] Boolean mask array. Must have the same size as *a*.

**vals**

[1-D sequence] Values to put into *a*. Only the first N elements are used, where N is the number of True values in *mask*. If *vals* is smaller than N, it will be repeated, and if elements of *a* are to be masked, this sequence must be non-empty.

**See also:**

`copyto`, `put`, `take`, `extract`

### Examples

```
>>> arr = np.arange(6).reshape(2, 3)
>>> np.place(arr, arr>2, [44, 55])
>>> arr
array([[ 0,  1,  2],
       [44, 55, 44]])
```

`numpy.put` (*a*, *ind*, *v*, *mode='raise'*)

Replaces specified elements of an array with given values.

The indexing works on the flattened target array. `put` is roughly equivalent to:

```
a.flat[ind] = v
```

**Parameters**

**a**

[ndarray] Target array.

**ind**

[array_like] Target indices, interpreted as integers.

**v**

[array_like] Values to place in *a* at target indices. If *v* is shorter than *ind* it will be repeated as necessary.

**mode**

[{'raise', 'wrap', 'clip'}, optional] Specifies how out-of-bounds indices will behave.

- 'raise' – raise an error (default)
- 'wrap' – wrap around
- 'clip' – clip to the range

'clip' mode means that all indices that are too large are replaced by the index that addresses the last element along that axis. Note that this disables indexing with negative numbers. In 'raise' mode, if an exception occurs the target array may still be modified.

**See also:**

`putmask`, `place`

**`put_along_axis`**

Put elements by matching the array and the index arrays

### Examples

```
>>> a = np.arange(5)
>>> np.put(a, [0, 2], [-44, -55])
>>> a
array([-44,   1, -55,   3,   4])
```

```
>>> a = np.arange(5)
>>> np.put(a, 22, -5, mode='clip')
>>> a
array([ 0,  1,  2,  3, -5])
```

`numpy.`**`put_along_axis`**(*arr*, *indices*, *values*, *axis*)

Put values into the destination array by matching 1d index and data slices.

This iterates over matching 1d slices oriented along the specified axis in the index and data arrays, and uses the former to place values into the latter. These slices can be different lengths.

Functions returning an index along an axis, like *`argsort`* and *`argpartition`*, produce suitable indices for this function.

New in version 1.15.0.

**Parameters**

**arr: ndarray (Ni…, M, Nk…)**

Destination array.

**indices: ndarray (Ni…, J, Nk…)**

Indices to change along each 1d slice of *arr*. This must match the dimension of arr, but dimensions in Ni and Nj may be 1 to broadcast against *arr*.

**values: array_like (Ni…, J, Nk…)**

values to insert at those indices. Its shape and dimension are broadcast to match that of *`indices`*.

**axis: int**

The axis to take 1d slices along. If axis is None, the destination array is treated as if a flattened 1d view had been created of it.

**See also:**

***`take_along_axis`***

Take values from the input array by matching 1d index and data slices

### Notes

This is equivalent to (but faster than) the following use of *ndindex* and *s_*, which sets each of `ii` and `kk` to a tuple of indices:

```
Ni, M, Nk = a.shape[:axis], a.shape[axis], a.shape[axis+1:]
J = indices.shape[axis]   # Need not equal M

for ii in ndindex(Ni):
    for kk in ndindex(Nk):
        a_1d       = a       [ii + s_[:,] + kk]
        indices_1d = indices[ii + s_[:,] + kk]
        values_1d  = values [ii + s_[:,] + kk]
        for j in range(J):
            a_1d[indices_1d[j]] = values_1d[j]
```

Equivalently, eliminating the inner loop, the last two lines would be:

```
a_1d[indices_1d] = values_1d
```

### Examples

For this sample array

```
>>> a = np.array([[10, 30, 20], [60, 40, 50]])
```

We can replace the maximum values with:

```
>>> ai = np.expand_dims(np.argmax(a, axis=1), axis=1)
>>> ai
array([[1],
       [0]])
>>> np.put_along_axis(a, ai, 99, axis=1)
>>> a
array([[10, 99, 20],
       [99, 40, 50]])
```

`numpy.`**`putmask`**(*a*, *mask*, *values*)

Changes elements of an array based on conditional and input values.

Sets `a.flat[n] = values[n]` for each n where `mask.flat[n]==True`.

If *values* is not the same size as *a* and *mask* then it will repeat. This gives behavior different from `a[mask] = values`.

**Parameters**

**a**

[array_like] Target array.

**mask**

[array_like] Boolean mask array. It has to be the same shape as *a*.

**values**

[array_like] Values to put into *a* where *mask* is True. If *values* is smaller than *a* it will be repeated.

**See also:**

*place*, *put*, *take*, *copyto*

## Examples

```
>>> x = np.arange(6).reshape(2, 3)
>>> np.putmask(x, x>2, x**2)
>>> x
array([[ 0,  1,  2],
       [ 9, 16, 25]])
```

If *values* is smaller than *a* it is repeated:

```
>>> x = np.arange(5)
>>> np.putmask(x, x>1, [-33, -44])
>>> x
array([  0,   1, -33, -44, -33])
```

`numpy.`**`fill_diagonal`**`(`*a*, *val*, *wrap=False*`)`

Fill the main diagonal of the given array of any dimensionality.

For an array *a* with `a.ndim >= 2`, the diagonal is the list of locations with indices `a[i, ..., i]` all identical. This function modifies the input array in-place, it does not return a value.

**Parameters**

**a**

[array, at least 2-D.] Array whose diagonal is to be filled, it gets modified in-place.

**val**

[scalar] Value to be written on the diagonal, its type must be compatible with that of the array a.

**wrap**

[bool] For tall matrices in NumPy version up to 1.6.2, the diagonal “wrapped” after N columns. You can have this behavior with this option. This affects only tall matrices.

**See also:**

*diag_indices*, *diag_indices_from*

## Notes

New in version 1.4.0.

This functionality can be obtained via *diag_indices*, but internally this version uses a much faster implementation that never constructs the indices and uses simple slicing.

### Examples

```
>>> a = np.zeros((3, 3), int)
>>> np.fill_diagonal(a, 5)
>>> a
array([[5, 0, 0],
       [0, 5, 0],
       [0, 0, 5]])
```

The same function can operate on a 4-D array:

```
>>> a = np.zeros((3, 3, 3, 3), int)
>>> np.fill_diagonal(a, 4)
```

We only show a few blocks for clarity:

```
>>> a[0, 0]
array([[4, 0, 0],
       [0, 0, 0],
       [0, 0, 0]])
>>> a[1, 1]
array([[0, 0, 0],
       [0, 4, 0],
       [0, 0, 0]])
>>> a[2, 2]
array([[0, 0, 0],
       [0, 0, 0],
       [0, 0, 4]])
```

The wrap option affects only tall matrices:

```
>>> # tall matrices no wrap
>>> a = np.zeros((5, 3), int)
>>> np.fill_diagonal(a, 4)
>>> a
array([[4, 0, 0],
       [0, 4, 0],
       [0, 0, 4],
       [0, 0, 0],
       [0, 0, 0]])
```

```
>>> # tall matrices wrap
>>> a = np.zeros((5, 3), int)
>>> np.fill_diagonal(a, 4, wrap=True)
>>> a
array([[4, 0, 0],
       [0, 4, 0],
       [0, 0, 4],
       [0, 0, 0],
       [4, 0, 0]])
```

```
>>> # wide matrices
>>> a = np.zeros((3, 5), int)
>>> np.fill_diagonal(a, 4, wrap=True)
>>> a
array([[4, 0, 0, 0, 0],
```

(continues on next page)

(continued from previous page)

```
        [0, 4, 0, 0, 0],
        [0, 0, 4, 0, 0]])
```

The anti-diagonal can be filled by reversing the order of elements using either `numpy.flipud` or `numpy.fliplr`.

```
>>> a = np.zeros((3, 3), int);
>>> np.fill_diagonal(np.fliplr(a), [1,2,3])  # Horizontal flip
>>> a
array([[0, 0, 1],
       [0, 2, 0],
       [3, 0, 0]])
>>> np.fill_diagonal(np.flipud(a), [1,2,3])  # Vertical flip
>>> a
array([[0, 0, 3],
       [0, 2, 0],
       [1, 0, 0]])
```

Note that the order in which the diagonal is filled varies depending on the flip function.

## 4.15.4 Iterating over arrays

| | |
|---|---|
| `nditer`(op[, flags, op_flags, op_dtypes, …]) | Efficient multi-dimensional iterator object to iterate over arrays. |
| `ndenumerate`(arr) | Multidimensional index iterator. |
| `ndindex`(*shape) | An N-dimensional iterator object to index arrays. |
| `nested_iters`() | Create nditers for use in nested loops |
| `flatiter` | Flat iterator object to iterate over arrays. |
| `lib.Arrayterator`(var[, buf_size]) | Buffered iterator for big arrays. |

**class** `numpy.`**`nditer`**(*op*, *flags=None*, *op_flags=None*, *op_dtypes=None*, *order='K'*, *casting='safe'*, *op_axes=None*, *itershape=None*, *buffersize=0*)

Efficient multi-dimensional iterator object to iterate over arrays. To get started using this object, see the *introductory guide to array iteration*.

**Parameters**

**op**

[ndarray or sequence of array_like] The array(s) to iterate over.

**flags**

[sequence of str, optional] Flags to control the behavior of the iterator.

- `buffered` enables buffering when required.
- `c_index` causes a C-order index to be tracked.
- `f_index` causes a Fortran-order index to be tracked.
- `multi_index` causes a multi-index, or a tuple of indices with one per iteration dimension, to be tracked.
- `common_dtype` causes all the operands to be converted to a common data type, with copying or buffering as necessary.

- `copy_if_overlap` causes the iterator to determine if read operands have overlap with write operands, and make temporary copies as necessary to avoid overlap. False positives (needless copying) are possible in some cases.
- `delay_bufalloc` delays allocation of the buffers until a reset() call is made. Allows `allocate` operands to be initialized before their values are copied into the buffers.
- `external_loop` causes the `values` given to be one-dimensional arrays with multiple values instead of zero-dimensional arrays.
- `grow_inner` allows the `value` array sizes to be made larger than the buffer size when both `buffered` and `external_loop` is used.
- `ranged` allows the iterator to be restricted to a sub-range of the iterindex values.
- `refs_ok` enables iteration of reference types, such as object arrays.
- `reduce_ok` enables iteration of `readwrite` operands which are broadcasted, also known as reduction operands.
- `zerosize_ok` allows `itersize` to be zero.

**op_flags**

[list of list of str, optional] This is a list of flags for each operand. At minimum, one of `readonly`, `readwrite`, or `writeonly` must be specified.

- `readonly` indicates the operand will only be read from.
- `readwrite` indicates the operand will be read from and written to.
- `writeonly` indicates the operand will only be written to.
- `no_broadcast` prevents the operand from being broadcasted.
- `contig` forces the operand data to be contiguous.
- `aligned` forces the operand data to be aligned.
- `nbo` forces the operand data to be in native byte order.
- `copy` allows a temporary read-only copy if required.
- `updateifcopy` allows a temporary read-write copy if required.
- `allocate` causes the array to be allocated if it is None in the `op` parameter.
- `no_subtype` prevents an `allocate` operand from using a subtype.
- `arraymask` indicates that this operand is the mask to use for selecting elements when writing to operands with the 'writemasked' flag set. The iterator does not enforce this, but when writing from a buffer back to the array, it only copies those elements indicated by this mask.
- `writemasked` indicates that only elements where the chosen `arraymask` operand is True will be written to.
- `overlap_assume_elementwise` can be used to mark operands that are accessed only in the iterator order, to allow less conservative copying when `copy_if_overlap` is present.

**op_dtypes**

[dtype or tuple of dtype(s), optional] The required data type(s) of the operands. If copying or buffering is enabled, the data will be converted to/from their original types.

**order**

[{‘C’, ‘F’, ‘A’, ‘K’}, optional] Controls the iteration order. ‘C’ means C order, ‘F’ means Fortran order, ‘A’ means ‘F’ order if all the arrays are Fortran contiguous, ‘C’ order otherwise, and ‘K’ means as close to the order the array elements appear in memory as possible. This also affects the element memory order of `allocate` operands, as they are allocated to be compatible with iteration order. Default is ‘K’.

**casting**

[{‘no’, ‘equiv’, ‘safe’, ‘same_kind’, ‘unsafe’}, optional] Controls what kind of data casting may occur when making a copy or buffering. Setting this to ‘unsafe’ is not recommended, as it can adversely affect accumulations.

- ‘no’ means the data types should not be cast at all.
- ‘equiv’ means only byte-order changes are allowed.
- ‘safe’ means only casts which can preserve values are allowed.
- ‘same_kind’ means only safe casts or casts within a kind, like float64 to float32, are allowed.
- ‘unsafe’ means any data conversions may be done.

**op_axes**

[list of list of ints, optional] If provided, is a list of ints or None for each operands. The list of axes for an operand is a mapping from the dimensions of the iterator to the dimensions of the operand. A value of -1 can be placed for entries, causing that dimension to be treated as *`newaxis`*.

**itershape**

[tuple of ints, optional] The desired shape of the iterator. This allows `allocate` operands with a dimension mapped by op_axes not corresponding to a dimension of a different operand to get a value not equal to 1 for that dimension.

**buffersize**

[int, optional] When buffering is enabled, controls the size of the temporary buffers. Set to 0 for the default value.

## Notes

*`nditer`* supersedes *`flatiter`*. The iterator implementation behind *`nditer`* is also exposed by the NumPy C API.

The Python exposure supplies two iteration interfaces, one which follows the Python iterator protocol, and another which mirrors the C-style do-while pattern. The native Python approach is better in most cases, but if you need the coordinates or index of an iterator, use the C-style pattern.

### Examples

Here is how we might write an `iter_add` function, using the Python iterator protocol:

```
>>> def iter_add_py(x, y, out=None):
...     addop = np.add
...     it = np.nditer([x, y, out], [],
...                 [['readonly'], ['readonly'], ['writeonly','allocate']])
...     with it:
...         for (a, b, c) in it:
...             addop(a, b, out=c)
...     return it.operands[2]
```

Here is the same function, but following the C-style pattern:

```
>>> def iter_add(x, y, out=None):
...    addop = np.add
...    it = np.nditer([x, y, out], [],
...                [['readonly'], ['readonly'], ['writeonly','allocate']])
...    with it:
...        while not it.finished:
...            addop(it[0], it[1], out=it[2])
...            it.iternext()
...        return it.operands[2]
```

Here is an example outer product function:

```
>>> def outer_it(x, y, out=None):
...     mulop = np.multiply
...     it = np.nditer([x, y, out], ['external_loop'],
...             [['readonly'], ['readonly'], ['writeonly', 'allocate']],
...             op_axes=[list(range(x.ndim)) + [-1] * y.ndim,
...                      [-1] * x.ndim + list(range(y.ndim)),
...                      None])
...     with it:
...         for (a, b, c) in it:
...             mulop(a, b, out=c)
...         return it.operands[2]
```

```
>>> a = np.arange(2)+1
>>> b = np.arange(3)+1
>>> outer_it(a,b)
array([[1, 2, 3],
       [2, 4, 6]])
```

Here is an example function which operates like a “lambda” ufunc:

```
>>> def luf(lamdaexpr, *args, **kwargs):
...    '''luf(lambdaexpr, op1, ..., opn, out=None, order='K', casting='safe',␣
↪buffersize=0)'''
...    nargs = len(args)
...    op = (kwargs.get('out',None),) + args
...    it = np.nditer(op, ['buffered','external_loop'],
...            [['writeonly','allocate','no_broadcast']] +
...                            [['readonly','nbo','aligned']]*nargs,
...            order=kwargs.get('order','K'),
...            casting=kwargs.get('casting','safe'),
```

(continues on next page)

(continued from previous page)

```
...             buffersize=kwargs.get('buffersize',0))
...     while not it.finished:
...         it[0] = lamdaexpr(*it[1:])
...         it.iternext()
...         return it.operands[0]
```

```
>>> a = np.arange(5)
>>> b = np.ones(5)
>>> luf(lambda i,j:i*i + j/2, a, b)
array([  0.5,    1.5,    4.5,    9.5,   16.5])
```

If operand flags *"writeonly"* or *"readwrite"* are used the operands may be views into the original data with the *WRITEBACKIFCOPY* flag. In this case `nditer` must be used as a context manager or the `nditer.close` method must be called before using the result. The temporary data will be written back to the original data when the `__exit__` function is called but not before:

```
>>> a = np.arange(6, dtype='i4')[::-2]
>>> with np.nditer(a, [],
...        [['writeonly', 'updateifcopy']],
...        casting='unsafe',
...        op_dtypes=[np.dtype('f4')]) as i:
...    x = i.operands[0]
...    x[:] = [-1, -2, -3]
...    # a still unchanged here
>>> a, x
(array([-1, -2, -3], dtype=int32), array([-1., -2., -3.], dtype=float32))
```

It is important to note that once the iterator is exited, dangling references (like *x* in the example) may or may not share data with the original data *a*. If writeback semantics were active, i.e. if *x.base.flags.writebackifcopy* is *True*, then exiting the iterator will sever the connection between *x* and *a*, writing to *x* will no longer write to *a*. If writeback semantics are not active, then *x.data* will still point at some part of *a.data*, and writing to one will affect the other.

Context management and the `close` method appeared in version 1.15.0.

**Attributes**

**dtypes**

[tuple of dtype(s)] The data types of the values provided in `value`. This may be different from the operand data types if buffering is enabled. Valid only before the iterator is closed.

**finished**

[bool] Whether the iteration over the operands is finished or not.

**has_delayed_bufalloc**

[bool] If True, the iterator was created with the `delay_bufalloc` flag, and no reset() function was called on it yet.

**has_index**

[bool] If True, the iterator was created with either the `c_index` or the `f_index` flag, and the property `index` can be used to retrieve it.

**has_multi_index**

[bool] If True, the iterator was created with the `multi_index` flag, and the property `multi_index` can be used to retrieve it.

**index**

When the `c_index` or `f_index` flag was used, this property provides access to the index. Raises a ValueError if accessed and `has_index` is False.

**iterationneedsapi**

[bool] Whether iteration requires access to the Python API, for example if one of the operands is an object array.

**iterindex**

[int] An index which matches the order of iteration.

**itersize**

[int] Size of the iterator.

**itviews**

Structured view(s) of `operands` in memory, matching the reordered and optimized iterator access pattern. Valid only before the iterator is closed.

**multi_index**

When the `multi_index` flag was used, this property provides access to the index. Raises a ValueError if accessed accessed and `has_multi_index` is False.

**ndim**

[int] The dimensions of the iterator.

**nop**

[int] The number of iterator operands.

**`operands`**

[tuple of operand(s)] operands[*Slice*]

**shape**

[tuple of ints] Shape tuple, the shape of the iterator.

**value**

Value of `operands` at current iteration. Normally, this is a tuple of array scalars, but if the flag `external_loop` is used, it is a tuple of one dimensional arrays.

## Methods

| | |
|---|---|
| `close`() | Resolve all writeback semantics in writeable operands. |
| `copy`() | Get a copy of the iterator in its current state. |
| `debug_print`() | Print the current state of the `nditer` instance and debug info to stdout. |
| `enable_external_loop`() | When the “external_loop” was not used during construction, but is desired, this modifies the iterator to behave as if the flag was specified. |
| `iternext`() | Check whether iterations are left, and perform a single internal iteration without returning the result. |
| `remove_axis`(i) | Removes axis *i* from the iterator. |

Continued on next page

Table 52 – continued from previous page

| | |
|---|---|
| *remove_multi_index*() | When the "multi_index" flag was specified, this removes it, allowing the internal iteration structure to be optimized further. |
| *reset*() | Reset the iterator to its initial state. |

method

`nditer.`**`close`**`()`

Resolve all writeback semantics in writeable operands.

New in version 1.15.0.

**See also:**

*Modifying Array Values*

method

`nditer.`**`copy`**`()`

Get a copy of the iterator in its current state.

### Examples

```
>>> x = np.arange(10)
>>> y = x + 1
>>> it = np.nditer([x, y])
>>> next(it)
(array(0), array(1))
>>> it2 = it.copy()
>>> next(it2)
(array(1), array(2))
```

method

`nditer.`**`debug_print`**`()`

Print the current state of the `nditer` instance and debug info to stdout.

method

`nditer.`**`enable_external_loop`**`()`

When the "external_loop" was not used during construction, but is desired, this modifies the iterator to behave as if the flag was specified.

method

`nditer.`**`iternext`**`()`

Check whether iterations are left, and perform a single internal iteration without returning the result. Used in the C-style pattern do-while pattern. For an example, see `nditer`.

**Returns**

**iternext**

[bool] Whether or not there are iterations left.

method

`nditer.`**`remove_axis`**`(`*i*`)`

Removes axis *i* from the iterator. Requires that the flag "multi_index" be enabled.

method

`nditer.`**`remove_multi_index`**`()`

When the "multi_index" flag was specified, this removes it, allowing the internal iteration structure to be optimized further.

method

`nditer.`**`reset`**`()`

Reset the iterator to its initial state.

**`class`** `numpy.`**`ndindex`**`(`**shape*`)`

An N-dimensional iterator object to index arrays.

Given the shape of an array, an *`ndindex`* instance iterates over the N-dimensional index of the array. At each iteration a tuple of indices is returned, the last dimension is iterated over first.

**Parameters**

**'*args'**

[ints] The size of each dimension of the array.

**See also:**

*`ndenumerate`*, *`flatiter`*

### Examples

```
>>> for index in np.ndindex(3, 2, 1):
...     print(index)
(0, 0, 0)
(0, 1, 0)
(1, 0, 0)
(1, 1, 0)
(2, 0, 0)
(2, 1, 0)
```

### Methods

| | |
|---|---|
| *`ndincr`*(self) | Increment the multi-dimensional index by one. |

method

`ndindex.`**`ndincr`**`(`*self*`)`

Increment the multi-dimensional index by one.

This method is for backward compatibility only: do not use.

`numpy.`**`nested_iters`**`()`

Create nditers for use in nested loops

Create a tuple of *`nditer`* objects which iterate in nested loops over different axes of the op argument. The first iterator is used in the outermost loop, the last in the innermost loop. Advancing one will change the subsequent iterators to point at its new element.

**Parameters**

**op**

[ndarray or sequence of array_like] The array(s) to iterate over.

**axes**

[list of list of int] Each item is used as an "op_axes" argument to an nditer

**flags, op_flags, op_dtypes, order, casting, buffersize (optional)**

See `nditer` parameters of the same name

**Returns**

**iters**

[tuple of nditer] An nditer for each item in *axes*, outermost first

**See also:**

`nditer`

### Examples

Basic usage. Note how y is the "flattened" version of [a[:, 0, :], a[:, 1, 0], a[:, 2, :]] since we specified the first iter's axes as [1]

```
>>> a = np.arange(12).reshape(2, 3, 2)
>>> i, j = np.nested_iters(a, [[1], [0, 2]], flags=["multi_index"])
>>> for x in i:
...      print(i.multi_index)
...      for y in j:
...          print('', j.multi_index, y)
(0,)
 (0, 0) 0
 (0, 1) 1
 (1, 0) 6
 (1, 1) 7
(1,)
 (0, 0) 2
 (0, 1) 3
 (1, 0) 8
 (1, 1) 9
(2,)
 (0, 0) 4
 (0, 1) 5
 (1, 0) 10
 (1, 1) 11
```

**class** numpy.**flatiter**

Flat iterator object to iterate over arrays.

A `flatiter` iterator is returned by `x.flat` for any array *x*. It allows iterating over the array as if it were a 1-D array, either in a for-loop or by calling its *next* method.

Iteration is done in row-major, C-style order (the last index varying the fastest). The iterator can also be indexed using basic slicing or advanced indexing.

**See also:**

***`ndarray.flat`***

Return a flat iterator over an array.

***`ndarray.flatten`***

Returns a flattened copy of an array.

### Notes

A *`flatiter`* iterator can not be constructed directly from Python code by calling the *`flatiter`* constructor.

### Examples

```
>>> x = np.arange(6).reshape(2, 3)
>>> fl = x.flat
>>> type(fl)
<class 'numpy.flatiter'>
>>> for item in fl:
...     print(item)
...
0
1
2
3
4
5
```

```
>>> fl[2:4]
array([2, 3])
```

**Attributes**

**`base`**

A reference to the array that is iterated over.

**`coords`**

An N-dimensional tuple of current coordinates.

**`index`**

Current flat index into the array.

### Methods

| | |
|---|---|
| *`copy`*() | Get a copy of the iterator as a 1-D array. |

method

`flatiter.`**`copy`**`()`

Get a copy of the iterator as a 1-D array.

### Examples

```
>>> x = np.arange(6).reshape(2, 3)
>>> x
array([[0, 1, 2],
       [3, 4, 5]])
>>> fl = x.flat
>>> fl.copy()
array([0, 1, 2, 3, 4, 5])
```

**class** `numpy.lib.`**`Arrayterator`**(*var*, *buf_size=None*)

Buffered iterator for big arrays.

*`Arrayterator`* creates a buffered iterator for reading big arrays in small contiguous blocks. The class is useful for objects stored in the file system. It allows iteration over the object *without* reading everything in memory; instead, small blocks are read and iterated over.

*`Arrayterator`* can be used with any object that supports multidimensional slices. This includes NumPy arrays, but also variables from Scientific.IO.NetCDF or pynetcdf for example.

**Parameters**

**var**

[array_like] The object to iterate over.

**buf_size**

[int, optional] The buffer size. If *buf_size* is supplied, the maximum amount of data that will be read into memory is *buf_size* elements. Default is None, which will read as many element as possible into memory.

**See also:**

**`ndenumerate`**

Multidimensional array iterator.

**`flatiter`**

Flat array iterator.

**`memmap`**

Create a memory-map to an array stored in a binary file on disk.

### Notes

The algorithm works by first finding a "running dimension", along which the blocks will be extracted. Given an array of dimensions `(d1, d2, ..., dn)`, e.g. if *buf_size* is smaller than `d1`, the first dimension will be used. If, on the other hand, `d1 < buf_size < d1*d2` the second dimension will be used, and so on. Blocks are extracted along this dimension, and when the last block is returned the process continues from the next dimension, until all elements have been read.

### Examples

```
>>> a = np.arange(3 * 4 * 5 * 6).reshape(3, 4, 5, 6)
>>> a_itor = np.lib.Arrayterator(a, 2)
>>> a_itor.shape
(3, 4, 5, 6)
```

Now we can iterate over `a_itor`, and it will return arrays of size two. Since *buf_size* was smaller than any dimension, the first dimension will be iterated over first:

```
>>> for subarr in a_itor:
...     if not subarr.all():
...         print(subarr, subarr.shape)
>>> # [[[[0 1]]]] (1, 1, 1, 2)
```

**Attributes**

**var**

**buf_size**

**start**

**stop**

**step**

**`shape`**

The shape of the array to be iterated over.

**`flat`**

A 1-D flat iterator for Arrayterator objects.

## 4.16 Input and output

### 4.16.1 NumPy binary files (NPY, NPZ)

| | |
|---|---|
| *`load`*(file[, mmap_mode, allow_pickle, …]) | Load arrays or pickled objects from `.npy`, `.npz` or pickled files. |
| *`save`*(file, arr[, allow_pickle, fix_imports]) | Save an array to a binary file in NumPy `.npy` format. |
| *`savez`*(file, *args, **kwds) | Save several arrays into a single file in uncompressed `.npz` format. |
| *`savez_compressed`*(file, *args, **kwds) | Save several arrays into a single file in compressed `.npz` format. |

`numpy.`**`load`**(*file*, *mmap_mode=None*, *allow_pickle=False*, *fix_imports=True*, *encoding='ASCII'*)

Load arrays or pickled objects from `.npy`, `.npz` or pickled files.

> **Warning:** Loading files that contain object arrays uses the `pickle` module, which is not secure against erroneous or maliciously constructed data. Consider passing `allow_pickle=False` to load data that is known not to contain object arrays for the safer handling of untrusted sources.

**Parameters**

**file**

[file-like object, string, or pathlib.Path] The file to read. File-like objects must support the `seek()` and `read()` methods. Pickled files require that the file-like object support the `readline()` method as well.

**mmap_mode**

[{None, 'r+', 'r', 'w+', 'c'}, optional] If not None, then memory-map the file, using the given mode (see `numpy.memmap` for a detailed description of the modes). A memory-mapped array is kept on disk. However, it can be accessed and sliced like any ndarray. Memory mapping is especially useful for accessing small fragments of large files without reading the entire file into memory.

**allow_pickle**

[bool, optional] Allow loading pickled object arrays stored in npy files. Reasons for disallowing pickles include security, as loading pickled data can execute arbitrary code. If pickles are disallowed, loading object arrays will fail. Default: False

Changed in version 1.16.3: Made default False in response to CVE-2019-6446.

**fix_imports**

[bool, optional] Only useful when loading Python 2 generated pickled files on Python 3, which includes npy/npz files containing object arrays. If *fix_imports* is True, pickle will try to map the old Python 2 names to the new names used in Python 3.

**encoding**

[str, optional] What encoding to use when reading Python 2 strings. Only useful when loading Python 2 generated pickled files in Python 3, which includes npy/npz files containing object arrays. Values other than 'latin1', 'ASCII', and 'bytes' are not allowed, as they can corrupt numerical data. Default: 'ASCII'

**Returns**

**result**

[array, tuple, dict, etc.] Data stored in the file. For `.npz` files, the returned instance of NpzFile class must be closed to avoid leaking file descriptors.

**Raises**

**IOError**

If the input file does not exist or cannot be read.

**ValueError**

The file contains an object array, but allow_pickle=False given.

**See also:**

`save`, `savez`, `savez_compressed`, `loadtxt`

**`memmap`**

Create a memory-map to an array stored in a file on disk.

**`lib.format.open_memmap`**

Create or load a memory-mapped `.npy` file.

## Notes

- If the file contains pickle data, then whatever object is stored in the pickle is returned.
- If the file is a `.npy` file, then a single array is returned.
- If the file is a `.npz` file, then a dictionary-like object is returned, containing `{filename: array}` key-value pairs, one for each file in the archive.
- If the file is a `.npz` file, the returned value supports the context manager protocol in a similar fashion to the open function:

```
with load('foo.npz') as data:
    a = data['a']
```

  The underlying file descriptor is closed when exiting the 'with' block.

## Examples

Store data to disk, and load it again:

```
>>> np.save('/tmp/123', np.array([[1, 2, 3], [4, 5, 6]]))
>>> np.load('/tmp/123.npy')
array([[1, 2, 3],
       [4, 5, 6]])
```

Store compressed data to disk, and load it again:

```
>>> a=np.array([[1, 2, 3], [4, 5, 6]])
>>> b=np.array([1, 2])
>>> np.savez('/tmp/123.npz', a=a, b=b)
>>> data = np.load('/tmp/123.npz')
>>> data['a']
array([[1, 2, 3],
       [4, 5, 6]])
>>> data['b']
array([1, 2])
>>> data.close()
```

Mem-map the stored array, and then access the second row directly from disk:

```
>>> X = np.load('/tmp/123.npy', mmap_mode='r')
>>> X[1, :]
memmap([4, 5, 6])
```

`numpy.`**`save`**(*file*, *arr*, *allow_pickle=True*, *fix_imports=True*)

Save an array to a binary file in NumPy `.npy` format.

**Parameters**

**file**

[file, str, or pathlib.Path] File or filename to which the data is saved. If file is a file-object, then the filename is unchanged. If file is a string or Path, a `.npy` extension will be appended to the filename if it does not already have one.

**arr**

[array_like] Array data to be saved.

**allow_pickle**

[bool, optional] Allow saving object arrays using Python pickles. Reasons for disallowing pickles include security (loading pickled data can execute arbitrary code) and portability (pickled objects may not be loadable on different Python installations, for example if the stored objects require libraries that are not available, and not all pickled data is compatible between Python 2 and Python 3). Default: True

**fix_imports**

[bool, optional] Only useful in forcing objects in object arrays on Python 3 to be pickled in a Python 2 compatible way. If *fix_imports* is True, pickle will try to map the new Python 3 names to the old module names used in Python 2, so that the pickle data stream is readable with Python 2.

**See also:**

**`savez`**

Save several arrays into a `.npz` archive

`savetxt`, `load`

## Notes

For a description of the `.npy` format, see `numpy.lib.format`.

Any data saved to the file is appended to the end of the file.

## Examples

```
>>> from tempfile import TemporaryFile
>>> outfile = TemporaryFile()
```

```
>>> x = np.arange(10)
>>> np.save(outfile, x)
```

```
>>> _ = outfile.seek(0) # Only needed here to simulate closing & reopening file
>>> np.load(outfile)
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
>>> with open('test.npy', 'wb') as f:
...     np.save(f, np.array([1, 2]))
...     np.save(f, np.array([1, 3]))
>>> with open('test.npy', 'rb') as f:
...     a = np.load(f)
...     b = np.load(f)
```

(continues on next page)

(continued from previous page)

```
>>> print(a, b)
# [1 2] [1 3]
```

`numpy.`**`savez`**(*file*, **args*, ***kwds*)

Save several arrays into a single file in uncompressed `.npz` format.

If arguments are passed in with no keywords, the corresponding variable names, in the `.npz` file, are 'arr_0', 'arr_1', etc. If keyword arguments are given, the corresponding variable names, in the `.npz` file will match the keyword names.

**Parameters**

**file**

[str or file] Either the filename (string) or an open file (file-like object) where the data will be saved. If file is a string or a Path, the `.npz` extension will be appended to the filename if it is not already there.

**args**

[Arguments, optional] Arrays to save to the file. Since it is not possible for Python to know the names of the arrays outside `savez`, the arrays will be saved with names "arr_0", "arr_1", and so on. These arguments can be any expression.

**kwds**

[Keyword arguments, optional] Arrays to save to the file. Arrays will be saved in the file with the keyword names.

**Returns**

**None**

**See also:**

**`save`**

Save a single array to a binary file in NumPy format.

**`savetxt`**

Save an array to a file as plain text.

**`savez_compressed`**

Save several arrays into a compressed `.npz` archive

## Notes

The `.npz` file format is a zipped archive of files named after the variables they contain. The archive is not compressed and each file in the archive contains one variable in `.npy` format. For a description of the `.npy` format, see `numpy.lib.format`.

When opening the saved `.npz` file with `load` a *NpzFile* object is returned. This is a dictionary-like object which can be queried for its list of arrays (with the `.files` attribute), and for the arrays themselves.

When saving dictionaries, the dictionary keys become filenames inside the ZIP archive. Therefore, keys should be valid filenames. E.g., avoid keys that begin with `/` or contain `.`.

## Examples

```
>>> from tempfile import TemporaryFile
>>> outfile = TemporaryFile()
>>> x = np.arange(10)
>>> y = np.sin(x)
```

Using `savez` with *args, the arrays are saved with default names.

```
>>> np.savez(outfile, x, y)
>>> _ = outfile.seek(0) # Only needed here to simulate closing & reopening file
>>> npzfile = np.load(outfile)
>>> npzfile.files
['arr_0', 'arr_1']
>>> npzfile['arr_0']
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Using `savez` with **kwds, the arrays are saved with the keyword names.

```
>>> outfile = TemporaryFile()
>>> np.savez(outfile, x=x, y=y)
>>> _ = outfile.seek(0)
>>> npzfile = np.load(outfile)
>>> sorted(npzfile.files)
['x', 'y']
>>> npzfile['x']
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

`numpy.`**`savez_compressed`**(*file*, **args*, ***kwds*)

Save several arrays into a single file in compressed `.npz` format.

If keyword arguments are given, then filenames are taken from the keywords. If arguments are passed in with no keywords, then stored filenames are arr_0, arr_1, etc.

**Parameters**

**file**

[str or file] Either the filename (string) or an open file (file-like object) where the data will be saved. If file is a string or a Path, the `.npz` extension will be appended to the filename if it is not already there.

**args**

[Arguments, optional] Arrays to save to the file. Since it is not possible for Python to know the names of the arrays outside `savez`, the arrays will be saved with names "arr_0", "arr_1", and so on. These arguments can be any expression.

**kwds**

[Keyword arguments, optional] Arrays to save to the file. Arrays will be saved in the file with the keyword names.

**Returns**

**None**

**See also:**

***numpy.save***

Save a single array to a binary file in NumPy format.

***numpy.savetxt***

Save an array to a file as plain text.

***numpy.savez***

Save several arrays into an uncompressed `.npz` file format

***numpy.load***

Load the files created by savez_compressed.

### Notes

The `.npz` file format is a zipped archive of files named after the variables they contain. The archive is compressed with `zipfile.ZIP_DEFLATED` and each file in the archive contains one variable in `.npy` format. For a description of the `.npy` format, see `numpy.lib.format`.

When opening the saved `.npz` file with `load` a *NpzFile* object is returned. This is a dictionary-like object which can be queried for its list of arrays (with the `.files` attribute), and for the arrays themselves.

### Examples

```
>>> test_array = np.random.rand(3, 2)
>>> test_vector = np.random.rand(4)
>>> np.savez_compressed('/tmp/123', a=test_array, b=test_vector)
>>> loaded = np.load('/tmp/123.npz')
>>> print(np.array_equal(test_array, loaded['a']))
True
>>> print(np.array_equal(test_vector, loaded['b']))
True
```

The format of these binary file types is documented in `numpy.lib.format`

## 4.16.2 Text files

| | |
|---|---|
| `loadtxt`(fname[, dtype, comments, delimiter, …]) | Load data from a text file. |
| `savetxt`(fname, X[, fmt, delimiter, newline, …]) | Save an array to a text file. |
| `genfromtxt`(fname[, dtype, comments, …]) | Load data from a text file, with missing values handled as specified. |
| `fromregex`(file, regexp, dtype[, encoding]) | Construct an array from a text file, using regular expression parsing. |
| `fromstring`(string[, dtype, count, sep]) | A new 1-D array initialized from text data in a string. |
| `ndarray.tofile`(fid[, sep, format]) | Write array to a file as text or binary (default). |
| `ndarray.tolist`() | Return the array as an `a.ndim`-levels deep nested list of Python scalars. |

`numpy.`**`savetxt`** (*fname, X, fmt='%.18e', delimiter=' ', newline='n', header='', footer='', comments='# ', encoding=None*)

Save an array to a text file.

**Parameters**

**fname**

[filename or file handle] If the filename ends in `.gz`, the file is automatically saved in compressed gzip format. `loadtxt` understands gzipped files transparently.

**X**

[1D or 2D array_like] Data to be saved to a text file.

**fmt**

[str or sequence of strs, optional] A single format (%10.5f), a sequence of formats, or a multi-format string, e.g. 'Iteration %d – %10.5f', in which case *delimiter* is ignored. For complex *X*, the legal options for *fmt* are:

- a single specifier, *fmt='%.4e'*, resulting in numbers formatted like *'(%s+%sj)' % (fmt, fmt)*
- a full string specifying every real and imaginary part, e.g. *' %.4e %+.4ej %.4e %+.4ej %.4e %+.4ej'* for 3 columns
- a list of specifiers, one per column - in this case, the real and imaginary part must have separate specifiers, e.g. *['%.3e + %.3ej', '(%.15e%+.15ej)']* for 2 columns

**delimiter**

[str, optional] String or character separating columns.

**newline**

[str, optional] String or character separating lines.

New in version 1.5.0.

**header**

[str, optional] String that will be written at the beginning of the file.

New in version 1.7.0.

**footer**

[str, optional] String that will be written at the end of the file.

New in version 1.7.0.

**comments**

[str, optional] String that will be prepended to the `header` and `footer` strings, to mark them as comments. Default: '# ', as expected by e.g. `numpy.loadtxt`.

New in version 1.7.0.

**encoding**

[{None, str}, optional] Encoding used to encode the outputfile. Does not apply to output streams. If the encoding is something other than 'bytes' or 'latin1' you will not be able to load the file in NumPy versions < 1.14. Default is 'latin1'.

New in version 1.14.0.

**See also:**

***`save`***

Save an array to a binary file in NumPy `.npy` format

***savez***

Save several arrays into an uncompressed `.npz` archive

***savez_compressed***

Save several arrays into a compressed `.npz` archive

### Notes

Further explanation of the *fmt* parameter (`%[flag]width[.precision]specifier`):

**flags:**

`-` : left justify

`+` : Forces to precede result with + or -.

`0` : Left pad the number with zeros instead of space (see width).

**width:**

Minimum number of characters to be printed. The value is not truncated if it has more characters.

**precision:**

- For integer specifiers (eg. `d,i,o,x`), the minimum number of digits.
- For `e, E` and `f` specifiers, the number of digits to print after the decimal point.
- For `g` and `G`, the maximum number of significant digits.
- For `s`, the maximum number of characters.

**specifiers:**

`c` : character

`d` or `i` : signed decimal integer

`e` or `E` : scientific notation with `e` or `E`.

`f` : decimal floating point

`g,G` : use the shorter of `e,E` or `f`

`o` : signed octal

`s` : string of characters

`u` : unsigned decimal integer

`x,X` : unsigned hexadecimal integer

This explanation of `fmt` is not complete, for an exhaustive specification see [1].

### References

[1]

### Examples

```
>>> x = y = z = np.arange(0.0,5.0,1.0)
>>> np.savetxt('test.out', x, delimiter=',')   # X is an array
>>> np.savetxt('test.out', (x,y,z))   # x,y,z equal sized 1D arrays
>>> np.savetxt('test.out', x, fmt='%1.4e')   # use exponential notation
```

numpy.**genfromtxt**(*fname*, *dtype=<class 'float'>*, *comments='#'*, *delimiter=None*, *skip_header=0*, *skip_footer=0*, *converters=None*, *missing_values=None*, *filling_values=None*, *usecols=None*, *names=None*, *excludelist=None*, deletechars=" !#$%&'()*+, -./:;<=>?@[\]^{|}~", *replace_space='_'*, *autostrip=False*, *case_sensitive=True*, *defaultfmt='f%i'*, *unpack=None*, *usemask=False*, *loose=True*, *invalid_raise=True*, *max_rows=None*, *encoding='bytes'*)

Load data from a text file, with missing values handled as specified.

Each line past the first *skip_header* lines is split at the *delimiter* character, and characters following the *comments* character are discarded.

**Parameters**

**fname**

[file, str, pathlib.Path, list of str, generator] File, filename, list, or generator to read. If the filename extension is *gz* or `bz2`, the file is first decompressed. Note that generators must return byte strings. The strings in a list or produced by a generator are treated as lines.

**dtype**

[dtype, optional] Data type of the resulting array. If None, the dtypes will be determined by the contents of each column, individually.

**comments**

[str, optional] The character used to indicate the start of a comment. All the characters occurring on a line after a comment are discarded

**delimiter**

[str, int, or sequence, optional] The string used to separate values. By default, any consecutive whitespaces act as delimiter. An integer or sequence of integers can also be provided as width(s) of each field.

**skiprows**

[int, optional] *skiprows* was removed in numpy 1.10. Please use *skip_header* instead.

**skip_header**

[int, optional] The number of lines to skip at the beginning of the file.

**skip_footer**

[int, optional] The number of lines to skip at the end of the file.

**converters**

[variable, optional] The set of functions that convert the data of a column to a value. The converters can also be used to provide a default value for missing data: `converters = {3: lambda s: float(s or 0)}`.

**missing**

[variable, optional] *missing* was removed in numpy 1.10. Please use *missing_values* instead.

**missing_values**

[variable, optional] The set of strings corresponding to missing data.

**filling_values**

[variable, optional] The set of values to be used as default when the data are missing.

**usecols**

[sequence, optional] Which columns to read, with 0 being the first. For example, `usecols = (1, 4, 5)` will extract the 2nd, 5th and 6th columns.

**names**

[{None, True, str, sequence}, optional] If *names* is True, the field names are read from the first line after the first *skip_header* lines. This line can optionally be proceeded by a comment delimiter. If *names* is a sequence or a single-string of comma-separated names, the names will be used to define the field names in a structured dtype. If *names* is None, the names of the dtype fields will be used, if any.

**excludelist**

[sequence, optional] A list of names to exclude. This list is appended to the default list ['return','file','print']. Excluded names are appended an underscore: for example, *file* would become *file_*.

**deletechars**

[str, optional] A string combining invalid characters that must be deleted from the names.

**defaultfmt**

[str, optional] A format used to define default field names, such as "f%i" or "f_%02i".

**autostrip**

[bool, optional] Whether to automatically strip white spaces from the variables.

**replace_space**

[char, optional] Character(s) used in replacement of white spaces in the variables names. By default, use a '_'.

**case_sensitive**

[{True, False, 'upper', 'lower'}, optional] If True, field names are case sensitive. If False or 'upper', field names are converted to upper case. If 'lower', field names are converted to lower case.

**unpack**

[bool, optional] If True, the returned array is transposed, so that arguments may be unpacked using `x, y, z = loadtxt(...)`

**usemask**

[bool, optional] If True, return a masked array. If False, return a regular array.

**loose**

[bool, optional] If True, do not raise errors for invalid values.

**invalid_raise**

[bool, optional] If True, an exception is raised if an inconsistency is detected in the number of columns. If False, a warning is emitted and the offending lines are skipped.

**max_rows**

[int, optional] The maximum number of rows to read. Must not be used with skip_footer at the same time. If given, the value must be at least 1. Default is to read the entire file.

New in version 1.10.0.

**encoding**

[str, optional] Encoding used to decode the inputfile. Does not apply when *fname* is a file object. The special value 'bytes' enables backward compatibility workarounds that ensure that you receive byte arrays when possible and passes latin1 encoded strings to converters. Override this value to receive unicode arrays and pass strings as input to converters. If set to None the system default is used. The default value is 'bytes'.

New in version 1.14.0.

**Returns**

**out**

[ndarray] Data read from the text file. If *usemask* is True, this is a masked array.

**See also:**

`numpy.loadtxt`

equivalent function when no data is missing.

### Notes

- When spaces are used as delimiters, or when no delimiter has been given as input, there should not be any missing data between two fields.
- When the variables are named (either by a flexible dtype or with *names*), there must not be any header in the file (else a ValueError exception is raised).
- Individual values are not stripped of spaces by default. When using a custom converter, make sure the function does remove spaces.

### References

[1]

### Examples

```
>>> from io import StringIO
>>> import numpy as np
```

Comma delimited file with mixed dtype

```
>>> s = StringIO(u"1,1.3,abcde")
>>> data = np.genfromtxt(s, dtype=[('myint','i8'),('myfloat','f8'),
... ('mystring','S5')], delimiter=",")
>>> data
array((1, 1.3, b'abcde'),
      dtype=[('myint', '<i8'), ('myfloat', '<f8'), ('mystring', 'S5')])
```

Using dtype = None

```
>>> _ = s.seek(0) # needed for StringIO example only
>>> data = np.genfromtxt(s, dtype=None,
... names = ['myint','myfloat','mystring'], delimiter=",")
>>> data
array((1, 1.3, b'abcde'),
      dtype=[('myint', '<i8'), ('myfloat', '<f8'), ('mystring', 'S5')])
```

Specifying dtype and names

```
>>> _ = s.seek(0)
>>> data = np.genfromtxt(s, dtype="i8,f8,S5",
... names=['myint','myfloat','mystring'], delimiter=",")
>>> data
array((1, 1.3, b'abcde'),
      dtype=[('myint', '<i8'), ('myfloat', '<f8'), ('mystring', 'S5')])
```

An example with fixed-width columns

```
>>> s = StringIO(u"11.3abcde")
>>> data = np.genfromtxt(s, dtype=None, names=['intvar','fltvar','strvar'],
...     delimiter=[1,3,5])
>>> data
array((1, 1.3, b'abcde'),
      dtype=[('intvar', '<i8'), ('fltvar', '<f8'), ('strvar', 'S5')])
```

An example to show comments

```
>>> f = StringIO('''
... text,# of chars
... hello world,11
... numpy,5''')
>>> np.genfromtxt(f, dtype='S12,S12', delimiter=',')
array([(b'text', b''), (b'hello world', b'11'), (b'numpy', b'5')],
  dtype=[('f0', 'S12'), ('f1', 'S12')])
```

`numpy.`**`fromregex`**(*file*, *regexp*, *dtype*, *encoding=None*)

Construct an array from a text file, using regular expression parsing.

The returned array is always a structured array, and is constructed from all matches of the regular expression in the file. Groups in the regular expression are converted to fields of the structured array.

**Parameters**

**file**

[str or file] Filename or file object to read.

**regexp**

[str or regexp] Regular expression used to parse the file. Groups in the regular expression correspond to fields in the dtype.

**dtype**

[dtype or list of dtypes] Dtype for the structured array.

**encoding**

[str, optional] Encoding used to decode the inputfile. Does not apply to input streams.

New in version 1.14.0.

**Returns**

**output**

[ndarray] The output array, containing the part of the content of *file* that was matched by *regexp*. *output* is always a structured array.

**Raises**

**TypeError**

When `dtype` is not a valid dtype for a structured array.

**See also:**

`fromstring`, `loadtxt`

## Notes

Dtypes for structured arrays can be specified in several forms, but all forms specify at least the data type and field name. For details see `doc.structured_arrays`.

## Examples

```
>>> f = open('test.dat', 'w')
>>> _ = f.write("1312 foo\n1534  bar\n444   qux")
>>> f.close()
```

```
>>> regexp = r"(\d+)\s+(...)"  # match [digits, whitespace, anything]
>>> output = np.fromregex('test.dat', regexp,
...                       [('num', np.int64), ('key', 'S3')])
>>> output
array([(1312, b'foo'), (1534, b'bar'), ( 444, b'qux')],
      dtype=[('num', '<i8'), ('key', 'S3')])
```

(continues on next page)

(continued from previous page)

```
>>> output['num']
array([1312, 1534,  444])
```

### 4.16.3 Raw binary files

| | |
|---|---|
| `fromfile`(file[, dtype, count, sep, offset]) | Construct an array from data in a text or binary file. |
| `ndarray.tofile`(fid[, sep, format]) | Write array to a file as text or binary (default). |

### 4.16.4 String formatting

| | |
|---|---|
| `array2string`(a[, max_line_width, precision, …]) | Return a string representation of an array. |
| `array_repr`(arr[, max_line_width, precision, …]) | Return the string representation of an array. |
| `array_str`(a[, max_line_width, precision, …]) | Return a string representation of the data in an array. |
| `format_float_positional`(x[, precision, …]) | Format a floating-point scalar as a decimal string in positional notation. |
| `format_float_scientific`(x[, precision, …]) | Format a floating-point scalar as a decimal string in scientific notation. |

`numpy.`**`array2string`**(*a*, *max_line_width=None*, *precision=None*, *suppress_small=None*, *separator=' '*, *prefix=''*, *style=<no value>*, *formatter=None*, *threshold=None*, *edgeitems=None*, *sign=None*, *floatmode=None*, *suffix=''*, *, *legacy=None*)

Return a string representation of an array.

**Parameters**

**a**

[array_like] Input array.

**max_line_width**

[int, optional] Inserts newlines if text is longer than *max_line_width*. Defaults to `numpy.get_printoptions()['linewidth']`.

**precision**

[int or None, optional] Floating point precision. Defaults to `numpy.get_printoptions()['precision']`.

**suppress_small**

[bool, optional] Represent numbers "very close" to zero as zero; default is False. Very close is defined by precision: if the precision is 8, e.g., numbers smaller (in absolute value) than 5e-9 are represented as zero. Defaults to `numpy.get_printoptions()['suppress']`.

**separator**

[str, optional] Inserted between elements.

**prefix**

[str, optional]

**suffix: str, optional**

The length of the prefix and suffix strings are used to respectively align and wrap the output. An array is typically printed as:

```
prefix + array2string(a) + suffix
```

The output is left-padded by the length of the prefix string, and wrapping is forced at the column `max_line_width - len(suffix)`. It should be noted that the content of prefix and suffix strings are not included in the output.

**style**

[_NoValue, optional] Has no effect, do not use.

Deprecated since version 1.14.0.

**formatter**

[dict of callables, optional] If not None, the keys should indicate the type(s) that the respective formatting function applies to. Callables should return a string. Types that are not specified (by their corresponding keys) are handled by the default formatters. Individual types for which a formatter can be set are:

- 'bool'
- 'int'
- 'timedelta' : a `numpy.timedelta64`
- 'datetime' : a `numpy.datetime64`
- 'float'
- 'longfloat' : 128-bit floats
- 'complexfloat'
- 'longcomplexfloat' : composed of two 128-bit floats
- 'void' : type `numpy.void`
- 'numpystr' : types `numpy.string_` and `numpy.unicode_`
- 'str' : all other strings

Other keys that can be used to set a group of types at once are:

- 'all' : sets all types
- 'int_kind' : sets 'int'
- 'float_kind' : sets 'float' and 'longfloat'
- 'complex_kind' : sets 'complexfloat' and 'longcomplexfloat'
- 'str_kind' : sets 'str' and 'numpystr'

**threshold**

[int, optional] Total number of array elements which trigger summarization rather than full repr. Defaults to `numpy.get_printoptions()['threshold']`.

**edgeitems**

[int, optional] Number of array items in summary at beginning and end of each dimension. Defaults to `numpy.get_printoptions()['edgeitems']`.

**sign**

[string, either '-', '+', or ' ', optional] Controls printing of the sign of floating-point types. If '+', always print the sign of positive values. If ' ', always prints a space (whitespace character) in the sign position of positive values. If '-', omit the sign character of positive values. Defaults to `numpy.get_printoptions()['sign']`.

**floatmode**

[str, optional] Controls the interpretation of the *precision* option for floating-point types. Defaults to `numpy.get_printoptions()['floatmode']`. Can take the following values:

- 'fixed': Always print exactly *precision* fractional digits, even if this would print more or fewer digits than necessary to specify the value uniquely.
- 'unique': Print the minimum number of fractional digits necessary to represent each value uniquely. Different elements may have a different number of digits. The value of the *precision* option is ignored.
- 'maxprec': Print at most *precision* fractional digits, but if an element can be uniquely represented with fewer digits only print it with that many.
- 'maxprec_equal': Print at most *precision* fractional digits, but if every element in the array can be uniquely represented with an equal number of fewer digits, use that many digits for all elements.

**legacy**

[string or *False*, optional] If set to the string *'1.13'* enables 1.13 legacy printing mode. This approximates numpy 1.13 print output by including a space in the sign position of floats and different behavior for 0d arrays. If set to *False*, disables legacy mode. Unrecognized strings will be ignored with a warning for forward compatibility.

New in version 1.14.0.

**Returns**

**array_str**

[str] String representation of the array.

**Raises**

**TypeError**

if a callable in `formatter` does not return a string.

**See also:**

*`array_str`*, *`array_repr`*, *`set_printoptions`*, *`get_printoptions`*

### Notes

If a formatter is specified for a certain type, the *precision* keyword is ignored for that type.

This is a very flexible function; *array_repr* and *array_str* are using *array2string* internally so keywords with the same name should work identically in all three functions.

### Examples

```
>>> x = np.array([1e-16,1,2,3])
>>> np.array2string(x, precision=2, separator=',',
...                       suppress_small=True)
'[0.,1.,2.,3.]'
```

```
>>> x  = np.arange(3.)
>>> np.array2string(x, formatter={'float_kind':lambda x: "%.2f" % x})
'[0.00 1.00 2.00]'
```

```
>>> x  = np.arange(3)
>>> np.array2string(x, formatter={'int':lambda x: hex(x)})
'[0x0 0x1 0x2]'
```

`numpy.`**`array_repr`**(*arr*, *max_line_width=None*, *precision=None*, *suppress_small=None*)

Return the string representation of an array.

**Parameters**

**arr**

[ndarray] Input array.

**max_line_width**

[int, optional] Inserts newlines if text is longer than *max_line_width*. Defaults to `numpy.get_printoptions()['linewidth']`.

**precision**

[int, optional] Floating point precision. Defaults to `numpy.get_printoptions()['precision']`.

**suppress_small**

[bool, optional] Represent numbers "very close" to zero as zero; default is False. Very close is defined by precision: if the precision is 8, e.g., numbers smaller (in absolute value) than 5e-9 are represented as zero. Defaults to `numpy.get_printoptions()['suppress']`.

**Returns**

**string**

[str] The string representation of an array.

**See also:**

*array_str*, *array2string*, *set_printoptions*

### Examples

```
>>> np.array_repr(np.array([1,2]))
'array([1, 2])'
>>> np.array_repr(np.ma.array([0.]))
'MaskedArray([0.])'
>>> np.array_repr(np.array([], np.int32))
'array([], dtype=int32)'
```

```
>>> x = np.array([1e-6, 4e-7, 2, 3])
>>> np.array_repr(x, precision=6, suppress_small=True)
'array([0.000001,  0.      ,  2.      ,  3.      ])'
```

`numpy.`**`array_str`**(*a*, *max_line_width=None*, *precision=None*, *suppress_small=None*)

Return a string representation of the data in an array.

The data in the array is returned as a single string. This function is similar to *array_repr*, the difference being that *array_repr* also returns information on the kind of array and its data type.

**Parameters**

**a**

[ndarray] Input array.

**max_line_width**

[int, optional] Inserts newlines if text is longer than *max_line_width*. Defaults to `numpy.get_printoptions()['linewidth']`.

**precision**

[int, optional] Floating point precision. Defaults to `numpy.get_printoptions()['precision']`.

**suppress_small**

[bool, optional] Represent numbers "very close" to zero as zero; default is False. Very close is defined by precision: if the precision is 8, e.g., numbers smaller (in absolute value) than 5e-9 are represented as zero. Defaults to `numpy.get_printoptions()['suppress']`.

**See also:**

*array2string*, *array_repr*, *set_printoptions*

### Examples

```
>>> np.array_str(np.arange(3))
'[0 1 2]'
```

`numpy.`**`format_float_positional`**(*x*, *precision=None*, *unique=True*, *fractional=True*, *trim='k'*, *sign=False*, *pad_left=None*, *pad_right=None*)

Format a floating-point scalar as a decimal string in positional notation.

Provides control over rounding, trimming and padding. Uses and assumes IEEE unbiased rounding. Uses the "Dragon4" algorithm.

**Parameters**

**x**

[python float or numpy floating scalar] Value to format.

**precision**

[non-negative integer or None, optional] Maximum number of digits to print. May be None if `unique` is *True*, but must be an integer if unique is *False*.

**unique**

[boolean, optional] If *True*, use a digit-generation strategy which gives the shortest representation which uniquely identifies the floating-point number from other values of the same type, by judicious rounding. If *precision* was omitted, print out all necessary digits, otherwise digit generation is cut off after *precision* digits and the remaining value is rounded. If *False*, digits are generated as if printing an infinite-precision value and stopping after *precision* digits, rounding the remaining value.

**fractional**

[boolean, optional] If *True*, the cutoff of *precision* digits refers to the total number of digits after the decimal point, including leading zeros. If *False*, *precision* refers to the total number of significant digits, before or after the decimal point, ignoring leading zeros.

**trim**

[one of 'k', '.', '0', '-', optional] Controls post-processing trimming of trailing digits, as follows:

- 'k' : keep trailing zeros, keep decimal point (no trimming)
- '.' : trim all trailing zeros, leave decimal point
- '0' : trim all but the zero before the decimal point. Insert the zero if it is missing.
- '-' : trim trailing zeros and any trailing decimal point

**sign**

[boolean, optional] Whether to show the sign for positive values.

**pad_left**

[non-negative integer, optional] Pad the left side of the string with whitespace until at least that many characters are to the left of the decimal point.

**pad_right**

[non-negative integer, optional] Pad the right side of the string with whitespace until at least that many characters are to the right of the decimal point.

**Returns**

**rep**

[string] The string representation of the floating point value

**See also:**

`format_float_scientific`

### Examples

```
>>> np.format_float_positional(np.float32(np.pi))
'3.1415927'
>>> np.format_float_positional(np.float16(np.pi))
'3.14'
>>> np.format_float_positional(np.float16(0.3))
'0.3'
>>> np.format_float_positional(np.float16(0.3), unique=False, precision=10)
'0.3000488281'
```

`numpy.`**`format_float_scientific`**`(`*x*, *precision=None*, *unique=True*, *trim='k'*, *sign=False*, *pad_left=None*, *exp_digits=None*`)`

Format a floating-point scalar as a decimal string in scientific notation.

Provides control over rounding, trimming and padding. Uses and assumes IEEE unbiased rounding. Uses the "Dragon4" algorithm.

**Parameters**

**x**

[python float or numpy floating scalar] Value to format.

**precision**

[non-negative integer or None, optional] Maximum number of digits to print. May be None if `unique` is *True*, but must be an integer if unique is *False*.

**unique**

[boolean, optional] If *True*, use a digit-generation strategy which gives the shortest representation which uniquely identifies the floating-point number from other values of the same type, by judicious rounding. If *precision* was omitted, print all necessary digits, otherwise digit generation is cut off after *precision* digits and the remaining value is rounded. If *False*, digits are generated as if printing an infinite-precision value and stopping after *precision* digits, rounding the remaining value.

**trim**

[one of 'k', '.', '0', '-', optional] Controls post-processing trimming of trailing digits, as follows:

- 'k' : keep trailing zeros, keep decimal point (no trimming)
- '.' : trim all trailing zeros, leave decimal point
- '0' : trim all but the zero before the decimal point. Insert the zero if it is missing.
- '-' : trim trailing zeros and any trailing decimal point

**sign**

[boolean, optional] Whether to show the sign for positive values.

**pad_left**

[non-negative integer, optional] Pad the left side of the string with whitespace until at least that many characters are to the left of the decimal point.

**exp_digits**

[non-negative integer, optional] Pad the exponent with zeros until it contains at least this many digits. If omitted, the exponent will be at least 2 digits.

**Returns**

**rep**

[string] The string representation of the floating point value

**See also:**

`format_float_positional`

### Examples

```
>>> np.format_float_scientific(np.float32(np.pi))
'3.1415927e+00'
>>> s = np.float32(1.23e24)
>>> np.format_float_scientific(s, unique=False, precision=15)
'1.230000071797338e+24'
>>> np.format_float_scientific(s, exp_digits=4)
'1.23e+0024'
```

## 4.16.5 Memory mapping files

| | |
|---|---|
| `memmap` | Create a memory-map to an array stored in a *binary* file on disk. |

## 4.16.6 Text formatting options

| | |
|---|---|
| `set_printoptions`([precision, threshold, …]) | Set printing options. |
| `get_printoptions`() | Return the current print options. |
| `set_string_function`(f[, repr]) | Set a Python function to be used when pretty printing arrays. |
| `printoptions`(*args, **kwargs) | Context manager for setting print options. |

numpy.**set_printoptions**(*precision=None, threshold=None, edgeitems=None, linewidth=None, suppress=None, nanstr=None, infstr=None, formatter=None, sign=None, floatmode=None, *, legacy=None*)

Set printing options.

These options determine the way floating point numbers, arrays and other NumPy objects are displayed.

**Parameters**

**precision**

[int or None, optional] Number of digits of precision for floating point output (default 8). May be None if *floatmode* is not *fixed*, to print as many digits as necessary to uniquely specify the value.

**threshold**

[int, optional] Total number of array elements which trigger summarization rather than full repr (default 1000). To always use the full repr without summarization, pass `sys.maxsize`.

**edgeitems**

[int, optional] Number of array items in summary at beginning and end of each dimension (default 3).

**linewidth**

[int, optional] The number of characters per line for the purpose of inserting line breaks (default 75).

**suppress**

[bool, optional] If True, always print floating point numbers using fixed point notation, in which case numbers equal to zero in the current precision will print as zero. If False, then scientific notation is used when absolute value of the smallest number is < 1e-4 or the ratio of the maximum absolute value to the minimum is > 1e3. The default is False.

**nanstr**

[str, optional] String representation of floating point not-a-number (default nan).

**infstr**

[str, optional] String representation of floating point infinity (default inf).

**sign**

[string, either '-', '+', or ' ', optional] Controls printing of the sign of floating-point types. If '+', always print the sign of positive values. If ' ', always prints a space (whitespace character) in the sign position of positive values. If '-', omit the sign character of positive values. (default '-')

**formatter**

[dict of callables, optional] If not None, the keys should indicate the type(s) that the respective formatting function applies to. Callables should return a string. Types that are not specified (by their corresponding keys) are handled by the default formatters. Individual types for which a formatter can be set are:

- 'bool'
- 'int'
- 'timedelta' : a `numpy.timedelta64`
- 'datetime' : a `numpy.datetime64`
- 'float'
- 'longfloat' : 128-bit floats
- 'complexfloat'
- 'longcomplexfloat' : composed of two 128-bit floats
- 'numpystr' : types `numpy.string_` and `numpy.unicode_`
- 'object' : *np.object_* arrays
- 'str' : all other strings

Other keys that can be used to set a group of types at once are:

- 'all' : sets all types
- 'int_kind' : sets 'int'
- 'float_kind' : sets 'float' and 'longfloat'

- 'complex_kind' : sets 'complexfloat' and 'longcomplexfloat'
- 'str_kind' : sets 'str' and 'numpystr'

**floatmode**

[str, optional] Controls the interpretation of the *precision* option for floating-point types. Can take the following values (default maxprec_equal):

- **'fixed': Always print exactly *precision* fractional digits,**

  even if this would print more or fewer digits than necessary to specify the value uniquely.
- **'unique': Print the minimum number of fractional digits necessary**

  to represent each value uniquely. Different elements may have a different number of digits. The value of the *precision* option is ignored.
- **'maxprec': Print at most *precision* fractional digits, but if**

  an element can be uniquely represented with fewer digits only print it with that many.
- **'maxprec_equal': Print at most *precision* fractional digits,**

  but if every element in the array can be uniquely represented with an equal number of fewer digits, use that many digits for all elements.

**legacy**

[string or *False*, optional] If set to the string *'1.13'* enables 1.13 legacy printing mode. This approximates numpy 1.13 print output by including a space in the sign position of floats and different behavior for 0d arrays. If set to *False*, disables legacy mode. Unrecognized strings will be ignored with a warning for forward compatibility.

New in version 1.14.0.

**See also:**

*`get_printoptions`*, *`printoptions`*, *`set_string_function`*, *`array2string`*

## Notes

`formatter` is always reset with a call to *`set_printoptions`*.

Use *`printoptions`* as a context manager to set the values temporarily.

## Examples

Floating point precision can be set:

```
>>> np.set_printoptions(precision=4)
>>> np.array([1.123456789])
[1.1235]
```

Long arrays can be summarised:

```
>>> np.set_printoptions(threshold=5)
>>> np.arange(10)
array([0, 1, 2, ..., 7, 8, 9])
```

Small results can be suppressed:

```
>>> eps = np.finfo(float).eps
>>> x = np.arange(4.)
>>> x**2 - (x + eps)**2
array([-4.9304e-32, -4.4409e-16,  0.0000e+00,  0.0000e+00])
>>> np.set_printoptions(suppress=True)
>>> x**2 - (x + eps)**2
array([-0., -0.,  0.,  0.])
```

A custom formatter can be used to display array elements as desired:

```
>>> np.set_printoptions(formatter={'all':lambda x: 'int: '+str(-x)})
>>> x = np.arange(3)
>>> x
array([int: 0, int: -1, int: -2])
>>> np.set_printoptions()  # formatter gets reset
>>> x
array([0, 1, 2])
```

To put back the default options, you can use:

```
>>> np.set_printoptions(edgeitems=3, infstr='inf',
... linewidth=75, nanstr='nan', precision=8,
... suppress=False, threshold=1000, formatter=None)
```

Also to temporarily override options, use `printoptions` as a context manager:

```
>>> with np.printoptions(precision=2, suppress=True, threshold=5):
...     np.linspace(0, 10, 10)
array([ 0.  ,  1.11,  2.22, ...,  7.78,  8.89, 10.  ])
```

`numpy.`**`get_printoptions`**`()`

Return the current print options.

**Returns**

**print_opts**

[dict] Dictionary of current print options with keys

- precision : int
- threshold : int
- edgeitems : int
- linewidth : int
- suppress : bool
- nanstr : str
- infstr : str
- formatter : dict of callables
- sign : str

For a full description of these options, see `set_printoptions`.

**See also:**

`set_printoptions`, `printoptions`, `set_string_function` `numpy.`**`set_string_function`**`(`*f*, *repr=True*`)`

Set a Python function to be used when pretty printing arrays.

**Parameters**

**f**

[function or None] Function to be used to pretty print arrays. The function should expect a single array argument and return a string of the representation of the array. If None, the function is reset to the default NumPy function to print arrays.

**repr**

[bool, optional] If True (default), the function for pretty printing (`__repr__`) is set, if False the function that returns the default string representation (`__str__`) is set.

**See also:**

*`set_printoptions`*, *`get_printoptions`*

## Examples

```
>>> def pprint(arr):
...     return 'HA! - What are you going to do now?'
...
>>> np.set_string_function(pprint)
>>> a = np.arange(10)
>>> a
HA! - What are you going to do now?
>>> _ = a
>>> # [0 1 2 3 4 5 6 7 8 9]
```

We can reset the function to the default:

```
>>> np.set_string_function(None)
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

*repr* affects either pretty printing or normal string representation. Note that `__repr__` is still affected by setting `__str__` because the width of each array element in the returned string becomes equal to the length of the result of `__str__()`.

```
>>> x = np.arange(4)
>>> np.set_string_function(lambda x:'random', repr=False)
>>> x.__str__()
'random'
>>> x.__repr__()
'array([0, 1, 2, 3])'
```

`numpy.`**`printoptions`**`(`**args*, ***kwargs*`)`

Context manager for setting print options.

Set print options for the scope of the *with* block, and restore the old options at the end. See *`set_printoptions`* for the full description of available options.

**See also:**

*`set_printoptions`*, *`get_printoptions`*

### Examples

```
>>> from numpy.testing import assert_equal
>>> with np.printoptions(precision=2):
...     np.array([2.0]) / 3
array([0.67])
```

The *as*-clause of the *with*-statement gives the current print options:

```
>>> with np.printoptions(precision=2) as opts:
...      assert_equal(opts, np.get_printoptions())
```

## 4.16.7 Base-n representations

| | |
|---|---|
| *binary_repr*(num[, width]) | Return the binary representation of the input number as a string. |
| *base_repr*(number[, base, padding]) | Return a string representation of a number in the given base system. |

numpy.**base_repr**(*number*, *base=2*, *padding=0*)

Return a string representation of a number in the given base system.

**Parameters**

**number**

[int] The value to convert. Positive and negative values are handled.

**base**

[int, optional] Convert `number` to the *base* number system. The valid range is 2-36, the default value is 2.

**padding**

[int, optional] Number of zeros padded on the left. Default is 0 (no padding).

**Returns**

**out**

[str] String representation of `number` in *base* system.

**See also:**

***binary_repr***

Faster version of *base_repr* for base 2.

### Examples

```
>>> np.base_repr(5)
'101'
>>> np.base_repr(6, 5)
'11'
>>> np.base_repr(7, base=5, padding=3)
'00012'
```

```
>>> np.base_repr(10, base=16)
'A'
>>> np.base_repr(32, base=16)
'20'
```

## 4.16.8 Data sources

| | |
|---|---|
| `DataSource`([destpath]) | A generic data source file (file, http, ftp, …). |

**class** numpy.**DataSource**(*destpath='.'*)

A generic data source file (file, http, ftp, …).

DataSources can be local files or remote files/URLs. The files may also be compressed or uncompressed. DataSource hides some of the low-level details of downloading the file, allowing you to simply pass in a valid file path (or URL) and obtain a file object.

**Parameters**

**destpath**

[str or None, optional] Path to the directory where the source file gets downloaded to for use. If *destpath* is None, a temporary directory will be created. The default path is the current directory.

### Notes

URLs require a scheme string (`http://`) to be used, without it they will fail:

```
>>> repos = np.DataSource()
>>> repos.exists('www.google.com/index.html')
False
>>> repos.exists('http://www.google.com/index.html')
True
```

Temporary directories are deleted when the DataSource is deleted.

### Examples

```
>>> ds = np.DataSource('/home/guido')
>>> urlname = 'http://www.google.com/'
>>> gfile = ds.open('http://www.google.com/')
>>> ds.abspath(urlname)
'/home/guido/www.google.com/index.html'

>>> ds = np.DataSource(None)  # use with temporary file
>>> ds.open('/home/guido/foobar.txt')
<open file '/home/guido.foobar.txt', mode 'r' at 0x91d4430>
>>> ds.abspath('/home/guido/foobar.txt')
'/tmp/.../home/guido/foobar.txt'
```

### Methods

| | |
|---|---|
| `abspath`(self, path) | Return absolute path of file in the DataSource directory. |
| `exists`(self, path) | Test if path exists. |
| `open`(self, path[, mode, encoding, newline]) | Open and return file-like object. |

method

`DataSource.`**`abspath`**(*self*, *path*)

Return absolute path of file in the DataSource directory.

If *path* is an URL, then `abspath` will return either the location the file exists locally or the location it would exist when opened using the `open` method.

**Parameters**

**path**

[str] Can be a local file or a remote URL.

**Returns**

**out**

[str] Complete path, including the `DataSource` destination directory.

### Notes

The functionality is based on `os.path.abspath`.

method

`DataSource.`**`exists`**(*self*, *path*)

Test if path exists.

Test if *path* exists as (and in this order):

- a local file.
- a remote URL that has been downloaded and stored locally in the `DataSource` directory.
- a remote URL that has not been downloaded, but is valid and accessible.

**Parameters**

**path**

[str] Can be a local file or a remote URL.

**Returns**

**out**

[bool] True if *path* exists.

### Notes

When *path* is an URL, `exists` will return True if it's either stored locally in the `DataSource` directory, or is a valid remote URL. `DataSource` does not discriminate between the two, the file is accessible if it exists in either location.

method

`DataSource.`**`open`**(*self*, *path*, *mode='r'*, *encoding=None*, *newline=None*)

Open and return file-like object.

If *path* is an URL, it will be downloaded, stored in the `DataSource` directory and opened from there.

**Parameters**

**path**

[str] Local file path or URL to open.

**mode**

[{'r', 'w', 'a'}, optional] Mode to open *path*. Mode 'r' for reading, 'w' for writing, 'a' to append. Available modes depend on the type of object specified by *path*. Default is 'r'.

**encoding**

[{None, str}, optional] Open text file with given encoding. The default encoding will be what `io.open` uses.

**newline**

[{None, str}, optional] Newline to use when reading text file.

**Returns**

**out**

[file object] File object.

### 4.16.9 Binary Format Description

| | |
|---|---|
| *`lib.format`* | Binary serialization |

Binary serialization

#### NPY format

A simple format for saving numpy arrays to disk with the full information about them.

The `.npy` format is the standard binary file format in NumPy for persisting a *single* arbitrary NumPy array on disk. The format stores all of the shape and dtype information necessary to reconstruct the array correctly even on another machine with a different architecture. The format is designed to be as simple as possible while achieving its limited goals.

The `.npz` format is the standard format for persisting *multiple* NumPy arrays on disk. A `.npz` file is a zip file containing multiple `.npy` files, one for each array.

#### Capabilities

- Can represent all NumPy arrays including nested record arrays and object arrays.
- Represents the data in its native binary form.
- Supports Fortran-contiguous arrays directly.
- Stores all of the necessary information to reconstruct the array including shape and dtype on a machine of a different architecture. Both little-endian and big-endian arrays are supported, and a file with little-endian numbers will yield a little-endian array on any machine reading the file. The types are described in terms of their actual sizes. For example, if a machine with a 64-bit C "long int" writes out an array with "long ints", a reading machine with 32-bit C "long ints" will yield an array with 64-bit integers.
- Is straightforward to reverse engineer. Datasets often live longer than the programs that created them. A competent developer should be able to create a solution in their preferred programming language to read most `.npy` files that he has been given without much documentation.
- Allows memory-mapping of the data. See *open_memmep*.
- Can be read from a filelike stream object instead of an actual file.
- Stores object arrays, i.e. arrays containing elements that are arbitrary Python objects. Files with object arrays are not to be mmapable, but can be read and written to disk.

#### Limitations

- Arbitrary subclasses of numpy.ndarray are not completely preserved. Subclasses will be accepted for writing, but only the array data will be written out. A regular numpy.ndarray object will be created upon reading the file.

**Warning:** Due to limitations in the interpretation of structured dtypes, dtypes with fields with empty names will have the names replaced by 'f0', 'f1', etc. Such arrays will not round-trip through the format entirely accurately. The data is intact; only the field names will differ. We are working on a fix for this. This fix will not require a change in the file format. The arrays with such structures can still be saved and restored, and the correct dtype may be restored by using the `loadedarray.view(correct_dtype)` method.

### File extensions

We recommend using the `.npy` and `.npz` extensions for files saved in this format. This is by no means a requirement; applications may wish to use these file formats but use an extension specific to the application. In the absence of an obvious alternative, however, we suggest using `.npy` and `.npz`.

### Version numbering

The version numbering of these formats is independent of NumPy version numbering. If the format is upgraded, the code in *numpy.io* will still be able to read and write Version 1.0 files.

### Format Version 1.0

The first 6 bytes are a magic string: exactly `\x93NUMPY`.

The next 1 byte is an unsigned byte: the major version number of the file format, e.g. `\x01`.

The next 1 byte is an unsigned byte: the minor version number of the file format, e.g. `\x00`. Note: the version of the file format is not tied to the version of the numpy package.

The next 2 bytes form a little-endian unsigned short int: the length of the header data HEADER_LEN.

The next HEADER_LEN bytes form the header data describing the array's format. It is an ASCII string which contains a Python literal expression of a dictionary. It is terminated by a newline (`\n`) and padded with spaces (`\x20`) to make the total of `len(magic string) + 2 + len(length) + HEADER_LEN` be evenly divisible by 64 for alignment purposes.

The dictionary contains three keys:

**"descr"**

> [dtype.descr] An object that can be passed as an argument to the *`numpy.dtype`* constructor to create the array's dtype.

**"fortran_order"**

> [bool] Whether the array data is Fortran-contiguous or not. Since Fortran-contiguous arrays are a common form of non-C-contiguity, we allow them to be written directly to disk for efficiency.

**"shape"**

> [tuple of int] The shape of the array.

For repeatability and readability, the dictionary keys are sorted in alphabetic order. This is for convenience only. A writer SHOULD implement this if possible. A reader MUST NOT depend on this.

Following the header comes the array data. If the dtype contains Python objects (i.e. `dtype.hasobject is True`), then the data is a Python pickle of the array. Otherwise the data is the contiguous (either C- or Fortran-, depending on `fortran_order`) bytes of the array. Consumers can figure out the number of bytes by multiplying the number of elements given by the shape (noting that `shape=()` means there is 1 element) by `dtype.itemsize`.

### Format Version 2.0

The version 1.0 format only allowed the array header to have a total size of 65535 bytes. This can be exceeded by structured arrays with a large number of columns. The version 2.0 format extends the header size to 4 GiB. *`numpy.save`* will automatically save in 2.0 format if the data requires it, else it will always use the more compatible 1.0 format.

The description of the fourth element of the header therefore has become: "The next 4 bytes form a little-endian unsigned int: the length of the header data HEADER_LEN."

### Format Version 3.0

This version replaces the ASCII string (which in practice was latin1) with a utf8-encoded string, so supports structured types with any unicode field names.

### Notes

The `.npy` format, including motivation for creating it and a comparison of alternatives, is described in the "npy-format" NEP, however details have evolved with time and this document is more current.

## 4.17 Linear algebra (`numpy.linalg`)

The NumPy linear algebra functions rely on BLAS and LAPACK to provide efficient low level implementations of standard linear algebra algorithms. Those libraries may be provided by NumPy itself using C versions of a subset of their reference implementations but, when possible, highly optimized libraries that take advantage of specialized processor functionality are preferred. Examples of such libraries are OpenBLAS, MKL (TM), and ATLAS. Because those libraries are multithreaded and processor dependent, environmental variables and external packages such as threadpoolctl may be needed to control the number of threads or specify the processor architecture.

The SciPy library also contains a `linalg` submodule, and there is overlap in the functionality provided by the SciPy and NumPy submodules. SciPy contains functions not found in *`numpy.linalg`*, such as functions related to LU decomposition and the Schur decomposition, multiple ways of calculating the pseudoinverse, and matrix transcendentals such as the matrix logarithm. Some functions that exist in both have augmented functionality in `scipy.linalg`. For example, `scipy.linalg.eig` can take a second matrix argument for solving generalized eigenvalue problems. Some functions in NumPy, however, have more flexible broadcasting options. For example, *`numpy.linalg.solve`* can handle "stacked" arrays, while `scipy.linalg.solve` accepts only a single square array as its first argument.

### 4.17.1 Matrix and vector products

| | |
|---|---|
| *`dot`*(a, b[, out]) | Dot product of two arrays. |
| *`linalg.multi_dot`*(arrays, *[, out]) | Compute the dot product of two or more arrays in a single function call, while automatically selecting the fastest evaluation order. |
| *`vdot`*(a, b) | Return the dot product of two vectors. |
| *`inner`*(a, b) | Inner product of two arrays. |
| *`outer`*(a, b[, out]) | Compute the outer product of two vectors. |
| *`matmul`*(x1, x2, /[, out, casting, order, …]) | Matrix product of two arrays. |
| *`tensordot`*(a, b[, axes]) | Compute tensor dot product along specified axes. |

Continued on next page

Table 65 – continued from previous page

| | |
|---|---|
| *einsum*(subscripts, *operands[, out, dtype, …]) | Evaluates the Einstein summation convention on the operands. |
| *einsum_path*(subscripts, *operands[, optimize]) | Evaluates the lowest cost contraction order for an einsum expression by considering the creation of intermediate arrays. |
| *linalg.matrix_power*(a, n) | Raise a square matrix to the (integer) power *n*. |
| *kron*(a, b) | Kronecker product of two arrays. |

numpy.**dot**(*a*, *b*, *out=None*)

Dot product of two arrays. Specifically,

- If both *a* and *b* are 1-D arrays, it is inner product of vectors (without complex conjugation).
- If both *a* and *b* are 2-D arrays, it is matrix multiplication, but using *matmul* or `a @ b` is preferred.
- If either *a* or *b* is 0-D (scalar), it is equivalent to *multiply* and using `numpy.multiply(a, b)` or `a * b` is preferred.
- If *a* is an N-D array and *b* is a 1-D array, it is a sum product over the last axis of *a* and *b*.
- If *a* is an N-D array and *b* is an M-D array (where `M>=2`), it is a sum product over the last axis of *a* and the second-to-last axis of *b*:

```
dot(a, b)[i,j,k,m] = sum(a[i,j,:] * b[k,:,m])
```

**Parameters**

**a**

[array_like] First argument.

**b**

[array_like] Second argument.

**out**

[ndarray, optional] Output argument. This must have the exact kind that would be returned if it was not used. In particular, it must have the right type, must be C-contiguous, and its dtype must be the dtype that would be returned for *dot(a,b)*. This is a performance feature. Therefore, if these conditions are not met, an exception is raised, instead of attempting to be flexible.

**Returns**

**output**

[ndarray] Returns the dot product of *a* and *b*. If *a* and *b* are both scalars or both 1-D arrays then a scalar is returned; otherwise an array is returned. If *out* is given, then it is returned.

**Raises**

**ValueError**

If the last dimension of *a* is not the same size as the second-to-last dimension of *b*.

**See also:**

***`vdot`***

Complex-conjugating dot product.

***`tensordot`***

Sum products over arbitrary axes.

***`einsum`***

Einstein summation convention.

***`matmul`***

'@' operator as method with out parameter.

### Examples

```
>>> np.dot(3, 4)
12
```

Neither argument is complex-conjugated:

```
>>> np.dot([2j, 3j], [2j, 3j])
(-13+0j)
```

For 2-D arrays it is the matrix product:

```
>>> a = [[1, 0], [0, 1]]
>>> b = [[4, 1], [2, 2]]
>>> np.dot(a, b)
array([[4, 1],
       [2, 2]])
```

```
>>> a = np.arange(3*4*5*6).reshape((3,4,5,6))
>>> b = np.arange(3*4*5*6)[::-1].reshape((5,4,6,3))
>>> np.dot(a, b)[2,3,2,1,2,2]
499128
>>> sum(a[2,3,2,:] * b[1,2,:,2])
499128
```

`numpy.linalg.`**`multi_dot`**(*arrays*, ***, *out=None*)

Compute the dot product of two or more arrays in a single function call, while automatically selecting the fastest evaluation order.

*`multi_dot`* chains *`numpy.dot`* and uses optimal parenthesization of the matrices [1] [2]. Depending on the shapes of the matrices, this can speed up the multiplication a lot.

If the first argument is 1-D it is treated as a row vector. If the last argument is 1-D it is treated as a column vector. The other arguments must be 2-D.

Think of *`multi_dot`* as:

```
def multi_dot(arrays): return functools.reduce(np.dot, arrays)
```

**Parameters**

**arrays**

[sequence of array_like] If the first argument is 1-D it is treated as row vector. If the last argument is 1-D it is treated as column vector. The other arguments must be 2-D.

**out**

[ndarray, optional] Output argument. This must have the exact kind that would be returned if it was not used. In particular, it must have the right type, must be C-contiguous, and its dtype must be the dtype that would be returned for *dot(a, b)*. This is a performance feature. Therefore, if these conditions are not met, an exception is raised, instead of attempting to be flexible.

New in version 1.19.0.

**Returns**

**output**

[ndarray] Returns the dot product of the supplied arrays.

**See also:**

**`dot`**

dot multiplication with two arguments.

## Notes

The cost for a matrix multiplication can be calculated with the following function:

```
def cost(A, B):
    return A.shape[0] * A.shape[1] * B.shape[1]
```

Assume we have three matrices $A_{10x100}, B_{100x5}, C_{5x50}$.

The costs for the two different parenthesizations are as follows:

```
cost((AB)C) = 10*100*5 + 10*5*50   = 5000 + 2500   = 7500
cost(A(BC)) = 10*100*50 + 100*5*50 = 50000 + 25000 = 75000
```

## References

[1], [2]

## Examples

`multi_dot` allows you to write:

```
>>> from numpy.linalg import multi_dot
>>> # Prepare some data
>>> A = np.random.random((10000, 100))
>>> B = np.random.random((100, 1000))
>>> C = np.random.random((1000, 5))
>>> D = np.random.random((5, 333))
>>> # the actual dot multiplication
>>> _ = multi_dot([A, B, C, D])
```

instead of:

```
>>> _ = np.dot(np.dot(np.dot(A, B), C), D)
>>> # or
>>> _ = A.dot(B).dot(C).dot(D)
```

`numpy.`**`vdot`**(*a*, *b*)

Return the dot product of two vectors.

The vdot(*a*, *b*) function handles complex numbers differently than dot(*a*, *b*). If the first argument is complex the complex conjugate of the first argument is used for the calculation of the dot product.

Note that `vdot` handles multidimensional arrays differently than `dot`: it does *not* perform a matrix product, but flattens input arguments to 1-D vectors first. Consequently, it should only be used for vectors.

**Parameters**

**a**

[array_like] If *a* is complex the complex conjugate is taken before calculation of the dot product.

**b**

[array_like] Second argument to the dot product.

**Returns**

**output**

[ndarray] Dot product of *a* and *b*. Can be an int, float, or complex depending on the types of *a* and *b*.

**See also:**

**`dot`**

Return the dot product without using the complex conjugate of the first argument.

## Examples

```
>>> a = np.array([1+2j,3+4j])
>>> b = np.array([5+6j,7+8j])
>>> np.vdot(a, b)
(70-8j)
>>> np.vdot(b, a)
(70+8j)
```

Note that higher-dimensional arrays are flattened!

```
>>> a = np.array([[1, 4], [5, 6]])
>>> b = np.array([[4, 1], [2, 2]])
>>> np.vdot(a, b)
30
>>> np.vdot(b, a)
30
>>> 1*4 + 4*1 + 5*2 + 6*2
30
```

`numpy.`**`inner`**(*a*, *b*)

Inner product of two arrays.

Ordinary inner product of vectors for 1-D arrays (without complex conjugation), in higher dimensions a sum product over the last axes.

**Parameters**

**a, b**

[array_like] If *a* and *b* are nonscalar, their last dimensions must match.

**Returns**

**out**

[ndarray] *out.shape = a.shape[:-1] + b.shape[:-1]*

**Raises**

**ValueError**

If the last dimension of *a* and *b* has different size.

**See also:**

*`tensordot`*

Sum products over arbitrary axes.

*`dot`*

Generalised matrix product, using second last dimension of *b*.

*`einsum`*

Einstein summation convention.

## Notes

For vectors (1-D arrays) it computes the ordinary inner-product:

```
np.inner(a, b) = sum(a[:]*b[:])
```

More generally, if *ndim(a) = r > 0* and *ndim(b) = s > 0*:

```
np.inner(a, b) = np.tensordot(a, b, axes=(-1,-1))
```

or explicitly:

```
np.inner(a, b)[i0,...,ir-1,j0,...,js-1]
     = sum(a[i0,...,ir-1,:]*b[j0,...,js-1,:])
```

In addition *a* or *b* may be scalars, in which case:

```
np.inner(a,b) = a*b
```

## Examples

Ordinary inner product for vectors:

```
>>> a = np.array([1,2,3])
>>> b = np.array([0,1,0])
>>> np.inner(a, b)
2
```

A multidimensional example:

```
>>> a = np.arange(24).reshape((2,3,4))
>>> b = np.arange(4)
>>> np.inner(a, b)
array([[ 14,  38,  62],
       [ 86, 110, 134]])
```

An example where *b* is a scalar:

```
>>> np.inner(np.eye(2), 7)
array([[7., 0.],
       [0., 7.]])
```

`numpy.`**`outer`**(*a*, *b*, *out=None*)

Compute the outer product of two vectors.

Given two vectors, `a = [a0, a1, ..., aM]` and `b = [b0, b1, ..., bN]`, the outer product [1] is:

```
[[a0*b0  a0*b1 ... a0*bN ]
 [a1*b0    .
 [ ...          .
 [aM*b0            aM*bN ]]
```

**Parameters**

**a**

[(M,) array_like] First input vector. Input is flattened if not already 1-dimensional.

**b**

[(N,) array_like] Second input vector. Input is flattened if not already 1-dimensional.

**out**

[(M, N) ndarray, optional] A location where the result is stored

New in version 1.9.0.

**Returns**

**out**

[(M, N) ndarray] `out[i, j] = a[i] * b[j]`

**See also:**

*`inner`*

***einsum***

`einsum('i,j->ij', a.ravel(), b.ravel())` is the equivalent.

***ufunc.outer***

A generalization to dimensions other than 1D and other operations. `np.multiply.outer(a.ravel(), b.ravel())` is the equivalent.

***tensordot***

`np.tensordot(a.ravel(), b.ravel(), axes=((), ()))` is the equivalent.

### References

[1]

### Examples

Make a (*very* coarse) grid for computing a Mandelbrot set:

```
>>> rl = np.outer(np.ones((5,)), np.linspace(-2, 2, 5))
>>> rl
array([[-2., -1.,  0.,  1.,  2.],
       [-2., -1.,  0.,  1.,  2.],
       [-2., -1.,  0.,  1.,  2.],
       [-2., -1.,  0.,  1.,  2.],
       [-2., -1.,  0.,  1.,  2.]])
>>> im = np.outer(1j*np.linspace(2, -2, 5), np.ones((5,)))
>>> im
array([[0.+2.j, 0.+2.j, 0.+2.j, 0.+2.j, 0.+2.j],
       [0.+1.j, 0.+1.j, 0.+1.j, 0.+1.j, 0.+1.j],
       [0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j],
       [0.-1.j, 0.-1.j, 0.-1.j, 0.-1.j, 0.-1.j],
       [0.-2.j, 0.-2.j, 0.-2.j, 0.-2.j, 0.-2.j]])
>>> grid = rl + im
>>> grid
array([[-2.+2.j, -1.+2.j,  0.+2.j,  1.+2.j,  2.+2.j],
       [-2.+1.j, -1.+1.j,  0.+1.j,  1.+1.j,  2.+1.j],
       [-2.+0.j, -1.+0.j,  0.+0.j,  1.+0.j,  2.+0.j],
       [-2.-1.j, -1.-1.j,  0.-1.j,  1.-1.j,  2.-1.j],
       [-2.-2.j, -1.-2.j,  0.-2.j,  1.-2.j,  2.-2.j]])
```

An example using a “vector” of letters:

```
>>> x = np.array(['a', 'b', 'c'], dtype=object)
>>> np.outer(x, [1, 2, 3])
array([['a', 'aa', 'aaa'],
       ['b', 'bb', 'bbb'],
       ['c', 'cc', 'ccc']], dtype=object)
```

`numpy.`**`matmul`** (*x1*, *x2*, */*, *out=None*, ***, *casting='same_kind'*, *order='K'*, *dtype=None*, *subok=True*[, *signature*, *extobj*]) **`= <ufunc 'matmul'>`**

Matrix product of two arrays.

**Parameters**

**x1, x2**

[array_like] Input arrays, scalars not allowed.

**out**

[ndarray, optional] A location into which the result is stored. If provided, it must have a shape that matches the signature *(n,k),(k,m)->(n,m)*. If not provided or None, a freshly-allocated array is returned.

****kwargs**

For other keyword-only arguments, see the *ufunc docs*.

New in version 1.16: Now handles ufunc kwargs

**Returns**

**y**

[ndarray] The matrix product of the inputs. This is a scalar only when both x1, x2 are 1-d vectors.

**Raises**

**ValueError**

If the last dimension of *a* is not the same size as the second-to-last dimension of *b*.

If a scalar value is passed in.

**See also:**

**`vdot`**

Complex-conjugating dot product.

**`tensordot`**

Sum products over arbitrary axes.

**`einsum`**

Einstein summation convention.

**`dot`**

alternative matrix product with different broadcasting rules.

## Notes

The behavior depends on the arguments in the following way.

- If both arguments are 2-D they are multiplied like conventional matrices.
- If either argument is N-D, N > 2, it is treated as a stack of matrices residing in the last two indexes and broadcast accordingly.
- If the first argument is 1-D, it is promoted to a matrix by prepending a 1 to its dimensions. After matrix multiplication the prepended 1 is removed.
- If the second argument is 1-D, it is promoted to a matrix by appending a 1 to its dimensions. After matrix multiplication the appended 1 is removed.

`matmul` differs from `dot` in two important ways:

- Multiplication by scalars is not allowed, use `*` instead.
- Stacks of matrices are broadcast together as if the matrices were elements, respecting the signature `(n,k),(k,m)->(n,m)`:

```
>>> a = np.ones([9, 5, 7, 4])
>>> c = np.ones([9, 5, 4, 3])
>>> np.dot(a, c).shape
(9, 5, 7, 9, 5, 3)
>>> np.matmul(a, c).shape
(9, 5, 7, 3)
>>> # n is 7, k is 4, m is 3
```

The matmul function implements the semantics of the @ operator introduced in Python 3.5 following PEP465.

### Examples

For 2-D arrays it is the matrix product:

```
>>> a = np.array([[1, 0],
...               [0, 1]])
>>> b = np.array([[4, 1],
...               [2, 2]])
>>> np.matmul(a, b)
array([[4, 1],
       [2, 2]])
```

For 2-D mixed with 1-D, the result is the usual.

```
>>> a = np.array([[1, 0],
...               [0, 1]])
>>> b = np.array([1, 2])
>>> np.matmul(a, b)
array([1, 2])
>>> np.matmul(b, a)
array([1, 2])
```

Broadcasting is conventional for stacks of arrays

```
>>> a = np.arange(2 * 2 * 4).reshape((2, 2, 4))
>>> b = np.arange(2 * 2 * 4).reshape((2, 4, 2))
>>> np.matmul(a,b).shape
(2, 2, 2)
>>> np.matmul(a, b)[0, 1, 1]
98
>>> sum(a[0, 1, :] * b[0 , :, 1])
98
```

Vector, vector returns the scalar inner product, but neither argument is complex-conjugated:

```
>>> np.matmul([2j, 3j], [2j, 3j])
(-13+0j)
```

Scalar multiplication raises an error.

```
>>> np.matmul([1,2], 3)
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
...
ValueError: matmul: Input operand 1 does not have enough dimensions ...
```

New in version 1.10.0.

`numpy.tensordot`(*a*, *b*, *axes=2*)

Compute tensor dot product along specified axes.

Given two tensors, *a* and *b*, and an array_like object containing two array_like objects, `(a_axes, b_axes)`, sum the products of *a*'s and *b*'s elements (components) over the axes specified by `a_axes` and `b_axes`. The third argument can be a single non-negative integer_like scalar, `N`; if it is such, then the last `N` dimensions of *a* and the first `N` dimensions of *b* are summed over.

**Parameters**

**a, b**

[array_like] Tensors to "dot".

**axes**

[int or (2,) array_like]

- integer_like If an int N, sum over the last N axes of *a* and the first N axes of *b* in order. The sizes of the corresponding axes must match.
- (2,) array_like Or, a list of axes to be summed over, first sequence applying to *a*, second to *b*. Both elements array_like must be of the same length.

**Returns**

**output**

[ndarray] The tensor dot product of the input.

**See also:**

*dot*, *einsum*

## Notes

**Three common use cases are:**

- `axes = 0` : tensor product $a \otimes b$
- `axes = 1` : tensor dot product $a \cdot b$
- `axes = 2` : (default) tensor double contraction $a : b$

When *axes* is integer_like, the sequence for evaluation will be: first the -Nth axis in *a* and 0th axis in *b*, and the -1th axis in *a* and Nth axis in *b* last.

When there is more than one axis to sum over - and they are not the last (first) axes of *a* (*b*) - the argument *axes* should consist of two sequences of the same length, with the first axis to sum over given first in both sequences, the second axis second, and so forth.

The shape of the result consists of the non-contracted axes of the first tensor, followed by the non-contracted axes of the second.

### Examples

A “traditional” example:

```
>>> a = np.arange(60.).reshape(3,4,5)
>>> b = np.arange(24.).reshape(4,3,2)
>>> c = np.tensordot(a,b, axes=([1,0],[0,1]))
>>> c.shape
(5, 2)
>>> c
array([[4400., 4730.],
       [4532., 4874.],
       [4664., 5018.],
       [4796., 5162.],
       [4928., 5306.]])
>>> # A slower but equivalent way of computing the same...
>>> d = np.zeros((5,2))
>>> for i in range(5):
...   for j in range(2):
...     for k in range(3):
...       for n in range(4):
...         d[i,j] += a[k,n,i] * b[n,k,j]
>>> c == d
array([[ True,  True],
       [ True,  True],
       [ True,  True],
       [ True,  True],
       [ True,  True]])
```

An extended example taking advantage of the overloading of + and *:

```
>>> a = np.array(range(1, 9))
>>> a.shape = (2, 2, 2)
>>> A = np.array(('a', 'b', 'c', 'd'), dtype=object)
>>> A.shape = (2, 2)
>>> a; A
array([[[1, 2],
        [3, 4]],
       [[5, 6],
        [7, 8]]])
array([['a', 'b'],
       ['c', 'd']], dtype=object)
```

```
>>> np.tensordot(a, A) # third argument default is 2 for double-contraction
array(['abbcccdddd', 'aaaaabbbbbbcccccccdddddddd'], dtype=object)
```

```
>>> np.tensordot(a, A, 1)
array([[['acc', 'bdd'],
        ['aaacccc', 'bbbdddd']],
       [['aaaaacccccc', 'bbbbbdddddd'],
        ['aaaaaaacccccccc', 'bbbbbbbdddddddd']]], dtype=object)
```

```
>>> np.tensordot(a, A, 0) # tensor product (result too long to incl.)
array([[[[['a', 'b'],
          ['c', 'd']],
          ...
```

```
>>> np.tensordot(a, A, (0, 1))
array([[['abbbbb', 'cddddd'],
        ['aabbbbbb', 'ccdddddd']],
       [['aaabbbbbbb', 'cccddddddd'],
        ['aaaabbbbbbbb', 'ccccdddddddd']]], dtype=object)
```

```
>>> np.tensordot(a, A, (2, 1))
array([[['abb', 'cdd'],
        ['aaabbbb', 'cccdddd']],
       [['aaaaabbbbbb', 'cccccdddddd'],
        ['aaaaaaabbbbbbbb', 'cccccccdddddddd']]], dtype=object)
```

```
>>> np.tensordot(a, A, ((0, 1), (0, 1)))
array(['abbbcccccddddddd', 'aabbbbccccccdddddddd'], dtype=object)
```

```
>>> np.tensordot(a, A, ((2, 1), (1, 0)))
array(['acccbbdddd', 'aaaaacccccccbbbbbbdddddddd'], dtype=object)
```

`numpy.`**`einsum`**(*subscripts*, **operands*, *out=None*, *dtype=None*, *order='K'*, *casting='safe'*, *optimize=False*)

Evaluates the Einstein summation convention on the operands.

Using the Einstein summation convention, many common multi-dimensional, linear algebraic array operations can be represented in a simple fashion. In *implicit* mode `einsum` computes these values.

In *explicit* mode, `einsum` provides further flexibility to compute other array operations that might not be considered classical Einstein summation operations, by disabling, or forcing summation over specified subscript labels.

See the notes and examples for clarification.

**Parameters**

**subscripts**

[str] Specifies the subscripts for summation as comma separated list of subscript labels. An implicit (classical Einstein summation) calculation is performed unless the explicit indicator '->' is included as well as subscript labels of the precise output form.

**operands**

[list of array_like] These are the arrays for the operation.

**out**

[ndarray, optional] If provided, the calculation is done into this array.

**dtype**

[{data-type, None}, optional] If provided, forces the calculation to use the data type specified. Note that you may have to also give a more liberal *casting* parameter to allow the conversions. Default is None.

**order**

[{'C', 'F', 'A', 'K'}, optional] Controls the memory layout of the output. 'C' means it should be C contiguous. 'F' means it should be Fortran contiguous, 'A' means it should be 'F' if the inputs are all 'F', 'C' otherwise. 'K' means it should be as close to the layout as the inputs as is possible, including arbitrarily permuted axes. Default is 'K'.

**casting**

[{'no', 'equiv', 'safe', 'same_kind', 'unsafe'}, optional] Controls what kind of data casting may occur. Setting this to 'unsafe' is not recommended, as it can adversely affect accumulations.

- 'no' means the data types should not be cast at all.
- 'equiv' means only byte-order changes are allowed.
- 'safe' means only casts which can preserve values are allowed.
- 'same_kind' means only safe casts or casts within a kind, like float64 to float32, are allowed.
- 'unsafe' means any data conversions may be done.

Default is 'safe'.

**optimize**

[{False, True, 'greedy', 'optimal'}, optional] Controls if intermediate optimization should occur. No optimization will occur if False and True will default to the 'greedy' algorithm. Also accepts an explicit contraction list from the `np.einsum_path` function. See `np.einsum_path` for more details. Defaults to False.

**Returns**

**output**

[ndarray] The calculation based on the Einstein summation convention.

**See also:**

*einsum_path*, *dot*, *inner*, *outer*, *tensordot*, *linalg.multi_dot*

## Notes

New in version 1.6.0.

The Einstein summation convention can be used to compute many multi-dimensional, linear algebraic array operations. *einsum* provides a succinct way of representing these.

A non-exhaustive list of these operations, which can be computed by *einsum*, is shown below along with examples:

- Trace of an array, *numpy.trace*.
- Return a diagonal, *numpy.diag*.
- Array axis summations, *numpy.sum*.
- Transpositions and permutations, *numpy.transpose*.
- Matrix multiplication and dot product, *numpy.matmul* *numpy.dot*.
- Vector inner and outer products, *numpy.inner* *numpy.outer*.
- Broadcasting, element-wise and scalar multiplication, *numpy.multiply*.
- Tensor contractions, *numpy.tensordot*.
- Chained array operations, in efficient calculation order, *numpy.einsum_path*.

The subscripts string is a comma-separated list of subscript labels, where each label refers to a dimension of the corresponding operand. Whenever a label is repeated it is summed, so `np.einsum('i,i', a, b)` is equivalent to *np.inner(a,b)*. If a label appears only once, it is not summed, so `np.einsum('i', a)` produces a view of `a` with no changes. A further example `np.einsum('ij,jk', a, b)` describes traditional matrix multiplication and is equivalent to *`np.matmul(a,b)`*. Repeated subscript labels in one operand take the diagonal. For example, `np.einsum('ii', a)` is equivalent to *`np.trace(a)`*.

In *implicit mode*, the chosen subscripts are important since the axes of the output are reordered alphabetically. This means that `np.einsum('ij', a)` doesn't affect a 2D array, while `np.einsum('ji', a)` takes its transpose. Additionally, `np.einsum('ij,jk', a, b)` returns a matrix multiplication, while, `np.einsum('ij,jh', a, b)` returns the transpose of the multiplication since subscript 'h' precedes subscript 'i'.

In *explicit mode* the output can be directly controlled by specifying output subscript labels. This requires the identifier '->' as well as the list of output subscript labels. This feature increases the flexibility of the function since summing can be disabled or forced when required. The call `np.einsum('i->', a)` is like *`np.sum(a, axis=-1)`*, and `np.einsum('ii->i', a)` is like *`np.diag(a)`*. The difference is that *`einsum`* does not allow broadcasting by default. Additionally `np.einsum('ij,jh->ih', a, b)` directly specifies the order of the output subscript labels and therefore returns matrix multiplication, unlike the example above in implicit mode.

To enable and control broadcasting, use an ellipsis. Default NumPy-style broadcasting is done by adding an ellipsis to the left of each term, like `np.einsum('...ii->...i', a)`. To take the trace along the first and last axes, you can do `np.einsum('i...i', a)`, or to do a matrix-matrix product with the left-most indices instead of rightmost, one can do `np.einsum('ij...,jk...->ik...', a, b)`.

When there is only one operand, no axes are summed, and no output parameter is provided, a view into the operand is returned instead of a new array. Thus, taking the diagonal as `np.einsum('ii->i', a)` produces a view (changed in version 1.10.0).

*`einsum`* also provides an alternative way to provide the subscripts and operands as `einsum(op0, sublist0, op1, sublist1, ..., [sublistout])`. If the output shape is not provided in this format *`einsum`* will be calculated in implicit mode, otherwise it will be performed explicitly. The examples below have corresponding *`einsum`* calls with the two parameter methods.

New in version 1.10.0.

Views returned from einsum are now writeable whenever the input array is writeable. For example, `np.einsum('ijk...->kji...', a)` will now have the same effect as *`np.swapaxes(a, 0, 2)`* and `np.einsum('ii->i', a)` will return a writeable view of the diagonal of a 2D array.

New in version 1.12.0.

Added the `optimize` argument which will optimize the contraction order of an einsum expression. For a contraction with three or more operands this can greatly increase the computational efficiency at the cost of a larger memory footprint during computation.

Typically a 'greedy' algorithm is applied which empirical tests have shown returns the optimal path in the majority of cases. In some cases 'optimal' will return the superlative path through a more expensive, exhaustive search. For iterative calculations it may be advisable to calculate the optimal path once and reuse that path by supplying it as an argument. An example is given below.

See *`numpy.einsum_path`* for more details.

### Examples

```
>>> a = np.arange(25).reshape(5,5)
>>> b = np.arange(5)
>>> c = np.arange(6).reshape(2,3)
```

Trace of a matrix:

```
>>> np.einsum('ii', a)
60
>>> np.einsum(a, [0,0])
60
>>> np.trace(a)
60
```

Extract the diagonal (requires explicit form):

```
>>> np.einsum('ii->i', a)
array([ 0,  6, 12, 18, 24])
>>> np.einsum(a, [0,0], [0])
array([ 0,  6, 12, 18, 24])
>>> np.diag(a)
array([ 0,  6, 12, 18, 24])
```

Sum over an axis (requires explicit form):

```
>>> np.einsum('ij->i', a)
array([ 10,  35,  60,  85, 110])
>>> np.einsum(a, [0,1], [0])
array([ 10,  35,  60,  85, 110])
>>> np.sum(a, axis=1)
array([ 10,  35,  60,  85, 110])
```

For higher dimensional arrays summing a single axis can be done with ellipsis:

```
>>> np.einsum('...j->...', a)
array([ 10,  35,  60,  85, 110])
>>> np.einsum(a, [Ellipsis,1], [Ellipsis])
array([ 10,  35,  60,  85, 110])
```

Compute a matrix transpose, or reorder any number of axes:

```
>>> np.einsum('ji', c)
array([[0, 3],
       [1, 4],
       [2, 5]])
>>> np.einsum('ij->ji', c)
array([[0, 3],
       [1, 4],
       [2, 5]])
>>> np.einsum(c, [1,0])
array([[0, 3],
       [1, 4],
       [2, 5]])
>>> np.transpose(c)
array([[0, 3],
       [1, 4],
       [2, 5]])
```

Vector inner products:

```
>>> np.einsum('i,i', b, b)
30
>>> np.einsum(b, [0], b, [0])
30
>>> np.inner(b,b)
30
```

Matrix vector multiplication:

```
>>> np.einsum('ij,j', a, b)
array([ 30,  80, 130, 180, 230])
>>> np.einsum(a, [0,1], b, [1])
array([ 30,  80, 130, 180, 230])
>>> np.dot(a, b)
array([ 30,  80, 130, 180, 230])
>>> np.einsum('...j,j', a, b)
array([ 30,  80, 130, 180, 230])
```

Broadcasting and scalar multiplication:

```
>>> np.einsum('..., ...', 3, c)
array([[ 0,  3,  6],
       [ 9, 12, 15]])
>>> np.einsum(',ij', 3, c)
array([[ 0,  3,  6],
       [ 9, 12, 15]])
>>> np.einsum(3, [Ellipsis], c, [Ellipsis])
array([[ 0,  3,  6],
       [ 9, 12, 15]])
>>> np.multiply(3, c)
array([[ 0,  3,  6],
       [ 9, 12, 15]])
```

Vector outer product:

```
>>> np.einsum('i,j', np.arange(2)+1, b)
array([[0, 1, 2, 3, 4],
       [0, 2, 4, 6, 8]])
>>> np.einsum(np.arange(2)+1, [0], b, [1])
array([[0, 1, 2, 3, 4],
       [0, 2, 4, 6, 8]])
>>> np.outer(np.arange(2)+1, b)
array([[0, 1, 2, 3, 4],
       [0, 2, 4, 6, 8]])
```

Tensor contraction:

```
>>> a = np.arange(60.).reshape(3,4,5)
>>> b = np.arange(24.).reshape(4,3,2)
>>> np.einsum('ijk,jil->kl', a, b)
array([[4400., 4730.],
       [4532., 4874.],
       [4664., 5018.],
       [4796., 5162.],
       [4928., 5306.]])
>>> np.einsum(a, [0,1,2], b, [1,0,3], [2,3])
```

(continues on next page)

(continued from previous page)

```
array([[4400., 4730.],
       [4532., 4874.],
       [4664., 5018.],
       [4796., 5162.],
       [4928., 5306.]])
>>> np.tensordot(a,b, axes=([1,0],[0,1]))
array([[4400., 4730.],
       [4532., 4874.],
       [4664., 5018.],
       [4796., 5162.],
       [4928., 5306.]])
```

Writeable returned arrays (since version 1.10.0):

```
>>> a = np.zeros((3, 3))
>>> np.einsum('ii->i', a)[:] = 1
>>> a
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
```

Example of ellipsis use:

```
>>> a = np.arange(6).reshape((3,2))
>>> b = np.arange(12).reshape((4,3))
>>> np.einsum('ki,jk->ij', a, b)
array([[10, 28, 46, 64],
       [13, 40, 67, 94]])
>>> np.einsum('ki,...k->i...', a, b)
array([[10, 28, 46, 64],
       [13, 40, 67, 94]])
>>> np.einsum('k...,jk', a, b)
array([[10, 28, 46, 64],
       [13, 40, 67, 94]])
```

Chained array operations. For more complicated contractions, speed ups might be achieved by repeatedly computing a ‘greedy’ path or pre-computing the ‘optimal’ path and repeatedly applying it, using an *`einsum_path`* insertion (since version 1.12.0). Performance improvements can be particularly significant with larger arrays:

```
>>> a = np.ones(64).reshape(2,4,8)
```

Basic *`einsum`*: ~1520ms (benchmarked on 3.1GHz Intel i5.)

```
>>> for iteration in range(500):
...     _ = np.einsum('ijk,ilm,njm,nlk,abc->',a,a,a,a,a)
```

Sub-optimal *`einsum`* (due to repeated path calculation time): ~330ms

```
>>> for iteration in range(500):
...     _ = np.einsum('ijk,ilm,njm,nlk,abc->',a,a,a,a,a, optimize='optimal')
```

Greedy *`einsum`* (faster optimal path approximation): ~160ms

```
>>> for iteration in range(500):
...     _ = np.einsum('ijk,ilm,njm,nlk,abc->',a,a,a,a,a, optimize='greedy')
```

Optimal *`einsum`* (best usage pattern in some use cases): ~110ms

```
>>> path = np.einsum_path('ijk,ilm,njm,nlk,abc->',a,a,a,a,a, optimize='optimal
↪')[0]
>>> for iteration in range(500):
...     _ = np.einsum('ijk,ilm,njm,nlk,abc->',a,a,a,a,a, optimize=path)
```

`numpy.`**`einsum_path`**(*subscripts*, **operands*, *optimize='greedy'*)

Evaluates the lowest cost contraction order for an einsum expression by considering the creation of intermediate arrays.

**Parameters**

**subscripts**

[str] Specifies the subscripts for summation.

***operands**

[list of array_like] These are the arrays for the operation.

**optimize**

[{bool, list, tuple, 'greedy', 'optimal'}] Choose the type of path. If a tuple is provided, the second argument is assumed to be the maximum intermediate size created. If only a single argument is provided the largest input or output array size is used as a maximum intermediate size.

- if a list is given that starts with `einsum_path`, uses this as the contraction path
- if False no optimization is taken
- if True defaults to the 'greedy' algorithm
- 'optimal' An algorithm that combinatorially explores all possible ways of contracting the listed tensors and choosest the least costly path. Scales exponentially with the number of terms in the contraction.
- 'greedy' An algorithm that chooses the best pair contraction at each step. Effectively, this algorithm searches the largest inner, Hadamard, and then outer products at each step. Scales cubically with the number of terms in the contraction. Equivalent to the 'optimal' path for most contractions.

Default is 'greedy'.

**Returns**

**path**

[list of tuples] A list representation of the einsum path.

**string_repr**

[str] A printable representation of the einsum path.

**See also:**

*`einsum`*, *`linalg.multi_dot`*

### Notes

The resulting path indicates which terms of the input contraction should be contracted first, the result of this contraction is then appended to the end of the contraction list. This list can then be iterated over until all intermediate contractions are complete.

### Examples

We can begin with a chain dot example. In this case, it is optimal to contract the `b` and `c` tensors first as represented by the first element of the path `(1, 2)`. The resulting tensor is added to the end of the contraction and the remaining contraction `(0, 1)` is then completed.

```
>>> np.random.seed(123)
>>> a = np.random.rand(2, 2)
>>> b = np.random.rand(2, 5)
>>> c = np.random.rand(5, 2)
>>> path_info = np.einsum_path('ij,jk,kl->il', a, b, c, optimize='greedy')
>>> print(path_info[0])
['einsum_path', (1, 2), (0, 1)]
>>> print(path_info[1])
  Complete contraction:  ij,jk,kl->il # may vary
         Naive scaling:  4
     Optimized scaling:  3
      Naive FLOP count:  1.600e+02
  Optimized FLOP count:  5.600e+01
   Theoretical speedup:  2.857
  Largest intermediate:  4.000e+00 elements
-------------------------------------------------------------------------
scaling                  current                                remaining
-------------------------------------------------------------------------
   3                   kl,jk->jl                                ij,jl->il
   3                   jl,ij->il                                   il->il
```

A more complex index transformation example.

```
>>> I = np.random.rand(10, 10, 10, 10)
>>> C = np.random.rand(10, 10)
>>> path_info = np.einsum_path('ea,fb,abcd,gc,hd->efgh', C, C, I, C, C,
...                            optimize='greedy')
```

```
>>> print(path_info[0])
['einsum_path', (0, 2), (0, 3), (0, 2), (0, 1)]
>>> print(path_info[1])
  Complete contraction:  ea,fb,abcd,gc,hd->efgh # may vary
         Naive scaling:  8
     Optimized scaling:  5
      Naive FLOP count:  8.000e+08
  Optimized FLOP count:  8.000e+05
   Theoretical speedup:  1000.000
  Largest intermediate:  1.000e+04 elements
--------------------------------------------------------------------------
scaling                  current                                remaining
--------------------------------------------------------------------------
   5               abcd,ea->bcde                      fb,gc,hd,bcde->efgh
   5               bcde,fb->cdef                         gc,hd,cdef->efgh
```

(continues on next page)

(continued from previous page)

```
  5               cdef,gc->defg                         hd,defg->efgh
  5               defg,hd->efgh                            efgh->efgh
```

`numpy.linalg.`**`matrix_power`**`(`*a*, *n*`)`

Raise a square matrix to the (integer) power *n*.

For positive integers *n*, the power is computed by repeated matrix squarings and matrix multiplications. If `n == 0`, the identity matrix of the same shape as M is returned. If `n < 0`, the inverse is computed and then raised to the `abs(n)`.

---

**Note:** Stacks of object matrices are not currently supported.

---

**Parameters**

**a**

[(…, M, M) array_like] Matrix to be "powered".

**n**

[int] The exponent can be any integer or long integer, positive, negative, or zero.

**Returns**

**a**n**

[(…, M, M) ndarray or matrix object] The return value is the same shape and type as *M*; if the exponent is positive or zero then the type of the elements is the same as those of *M*. If the exponent is negative the elements are floating-point.

**Raises**

**LinAlgError**

For matrices that are not square or that (for negative powers) cannot be inverted numerically.

## Examples

```
>>> from numpy.linalg import matrix_power
>>> i = np.array([[0, 1], [-1, 0]]) # matrix equiv. of the imaginary unit
>>> matrix_power(i, 3) # should = -i
array([[ 0, -1],
       [ 1,  0]])
>>> matrix_power(i, 0)
array([[1, 0],
       [0, 1]])
>>> matrix_power(i, -3) # should = 1/(-i) = i, but w/ f.p. elements
array([[ 0.,  1.],
       [-1.,  0.]])
```

Somewhat more sophisticated example

```
>>> q = np.zeros((4, 4))
>>> q[0:2, 0:2] = -i
>>> q[2:4, 2:4] = i
>>> q # one of the three quaternion units not equal to 1
array([[ 0., -1.,  0.,  0.],
       [ 1.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  1.],
       [ 0.,  0., -1.,  0.]])
>>> matrix_power(q, 2) # = -np.eye(4)
array([[-1.,  0.,  0.,  0.],
       [ 0., -1.,  0.,  0.],
       [ 0.,  0., -1.,  0.],
       [ 0.,  0.,  0., -1.]])
```

`numpy.`**`kron`**`(`*a*, *b*`)`

Kronecker product of two arrays.

Computes the Kronecker product, a composite array made of blocks of the second array scaled by the first.

**Parameters**

**a, b**

[array_like]

**Returns**

**out**

[ndarray]

**See also:**

***`outer`***

The outer product

## Notes

The function assumes that the number of dimensions of *a* and *b* are the same, if necessary prepending the smallest with ones. If *a.shape = (r0,r1,..,rN)* and *b.shape = (s0,s1,...,sN)*, the Kronecker product has shape *(r0*s0, r1*s1, ..., rN*SN)*. The elements are products of elements from *a* and *b*, organized explicitly by:

```
kron(a,b)[k0,k1,...,kN] = a[i0,i1,...,iN] * b[j0,j1,...,jN]
```

where:

```
kt = it * st + jt,  t = 0,...,N
```

In the common 2-D case (N=1), the block structure can be visualized:

```
[[ a[0,0]*b,   a[0,1]*b,  ... , a[0,-1]*b  ],
 [  ...                              ...   ],
 [ a[-1,0]*b,  a[-1,1]*b, ... , a[-1,-1]*b ]]
```

### Examples

```
>>> np.kron([1,10,100], [5,6,7])
array([  5,   6,   7, ..., 500, 600, 700])
>>> np.kron([5,6,7], [1,10,100])
array([  5,  50, 500, ...,   7,  70, 700])
```

```
>>> np.kron(np.eye(2), np.ones((2,2)))
array([[1.,  1.,  0.,  0.],
       [1.,  1.,  0.,  0.],
       [0.,  0.,  1.,  1.],
       [0.,  0.,  1.,  1.]])
```

```
>>> a = np.arange(100).reshape((2,5,2,5))
>>> b = np.arange(24).reshape((2,3,4))
>>> c = np.kron(a,b)
>>> c.shape
(2, 10, 6, 20)
>>> I = (1,3,0,2)
>>> J = (0,2,1)
>>> J1 = (0,) + J             # extend to ndim=4
>>> S1 = (1,) + b.shape
>>> K = tuple(np.array(I) * np.array(S1) + np.array(J1))
>>> c[K] == a[I]*b[J]
True
```

## 4.17.2 Decompositions

| | |
|---|---|
| `linalg.cholesky`(a) | Cholesky decomposition. |
| `linalg.qr`(a[, mode]) | Compute the qr factorization of a matrix. |
| `linalg.svd`(a[, full_matrices, compute_uv, …]) | Singular Value Decomposition. |

`numpy.linalg.`**`cholesky`**`(`*a*`)`

Cholesky decomposition.

Return the Cholesky decomposition, *L* * *L.H*, of the square matrix *a*, where *L* is lower-triangular and .H is the conjugate transpose operator (which is the ordinary transpose if *a* is real-valued). *a* must be Hermitian (symmetric if real-valued) and positive-definite. No checking is performed to verify whether *a* is Hermitian or not. In addition, only the lower-triangular and diagonal elements of *a* are used. Only *L* is actually returned.

**Parameters**

**a**

[(…, M, M) array_like] Hermitian (symmetric if all elements are real), positive-definite input matrix.

**Returns**

**L**

[(…, M, M) array_like] Upper or lower-triangular Cholesky factor of *a*. Returns a matrix object if *a* is a matrix object.

**Raises**

**LinAlgError**

If the decomposition fails, for example, if *a* is not positive-definite.

**See also:**

**`scipy.linalg.cholesky`**

Similar function in SciPy.

**`scipy.linalg.cholesky_banded`**

Cholesky decompose a banded Hermitian positive-definite matrix.

**`scipy.linalg.cho_factor`**

Cholesky decomposition of a matrix, to use in `scipy.linalg.cho_solve`.

## Notes

New in version 1.8.0.

Broadcasting rules apply, see the *`numpy.linalg`* documentation for details.

The Cholesky decomposition is often used as a fast way of solving

$$A\mathbf{x} = \mathbf{b}$$

(when $A$ is both Hermitian/symmetric and positive-definite).

First, we solve for **y** in

$$L\mathbf{y} = \mathbf{b},$$

and then for **x** in

$$L.H\mathbf{x} = \mathbf{y}.$$

## Examples

```
>>> A = np.array([[1,-2j],[2j,5]])
>>> A
array([[ 1.+0.j, -0.-2.j],
       [ 0.+2.j,  5.+0.j]])
>>> L = np.linalg.cholesky(A)
>>> L
array([[1.+0.j, 0.+0.j],
       [0.+2.j, 1.+0.j]])
>>> np.dot(L, L.T.conj()) # verify that L * L.H = A
array([[1.+0.j, 0.-2.j],
       [0.+2.j, 5.+0.j]])
>>> A = [[1,-2j],[2j,5]] # what happens if A is only array_like?
>>> np.linalg.cholesky(A) # an ndarray object is returned
array([[1.+0.j, 0.+0.j],
       [0.+2.j, 1.+0.j]])
```

(continues on next page)

(continued from previous page)

```
>>> # But a matrix object is returned if A is a matrix object
>>> np.linalg.cholesky(np.matrix(A))
matrix([[ 1.+0.j,  0.+0.j],
        [ 0.+2.j,  1.+0.j]])
```

`numpy.linalg.`**`qr`**`(`*a*, *mode='reduced'*`)`

Compute the qr factorization of a matrix.

Factor the matrix *a* as *qr*, where *q* is orthonormal and *r* is upper-triangular.

**Parameters**

**a**

[array_like, shape (M, N)] Matrix to be factored.

**mode**

[{'reduced', 'complete', 'r', 'raw'}, optional] If K = min(M, N), then

- 'reduced' : returns q, r with dimensions (M, K), (K, N) (default)
- 'complete' : returns q, r with dimensions (M, M), (M, N)
- 'r' : returns r only with dimensions (K, N)
- 'raw' : returns h, tau with dimensions (N, M), (K,)

The options 'reduced', 'complete, and 'raw' are new in numpy 1.8, see the notes for more information. The default is 'reduced', and to maintain backward compatibility with earlier versions of numpy both it and the old default 'full' can be omitted. Note that array h returned in 'raw' mode is transposed for calling Fortran. The 'economic' mode is deprecated. The modes 'full' and 'economic' may be passed using only the first letter for backwards compatibility, but all others must be spelled out. See the Notes for more explanation.

**Returns**

**q**

[ndarray of float or complex, optional] A matrix with orthonormal columns. When mode = 'complete' the result is an orthogonal/unitary matrix depending on whether or not a is real/complex. The determinant may be either +/- 1 in that case.

**r**

[ndarray of float or complex, optional] The upper-triangular matrix.

**(h, tau)**

[ndarrays of np.double or np.cdouble, optional] The array h contains the Householder reflectors that generate q along with r. The tau array contains scaling factors for the reflectors. In the deprecated 'economic' mode only h is returned.

**Raises**

**LinAlgError**

If factoring fails.

**See also:**

**`scipy.linalg.qr`**

Similar function in SciPy.

**`scipy.linalg.rq`**

Compute RQ decomposition of a matrix.

### Notes

This is an interface to the LAPACK routines `dgeqrf`, `zgeqrf`, `dorgqr`, and `zungqr`.

For more information on the qr factorization, see for example: https://en.wikipedia.org/wiki/QR_factorization

Subclasses of *ndarray* are preserved except for the 'raw' mode. So if *a* is of type *matrix*, all the return values will be matrices too.

New 'reduced', 'complete', and 'raw' options for mode were added in NumPy 1.8.0 and the old option 'full' was made an alias of 'reduced'. In addition the options 'full' and 'economic' were deprecated. Because 'full' was the previous default and 'reduced' is the new default, backward compatibility can be maintained by letting *mode* default. The 'raw' option was added so that LAPACK routines that can multiply arrays by q using the Householder reflectors can be used. Note that in this case the returned arrays are of type np.double or np.cdouble and the h array is transposed to be FORTRAN compatible. No routines using the 'raw' return are currently exposed by numpy, but some are available in lapack_lite and just await the necessary work.

### Examples

```
>>> a = np.random.randn(9, 6)
>>> q, r = np.linalg.qr(a)
>>> np.allclose(a, np.dot(q, r))  # a does equal qr
True
>>> r2 = np.linalg.qr(a, mode='r')
>>> np.allclose(r, r2)  # mode='r' returns the same r as mode='full'
True
```

Example illustrating a common use of `qr`: solving of least squares problems

What are the least-squares-best *m* and *y0* in `y = y0 + mx` for the following data: {(0,1), (1,0), (1,2), (2,1)}. (Graph the points and you'll see that it should be y0 = 0, m = 1.) The answer is provided by solving the over-determined matrix equation `Ax = b`, where:

```
A = array([[0, 1], [1, 1], [1, 1], [2, 1]])
x = array([[y0], [m]])
b = array([[1], [0], [2], [1]])
```

If A = qr such that q is orthonormal (which is always possible via Gram-Schmidt), then `x = inv(r) * (q.T) * b`. (In numpy practice, however, we simply use `lstsq`.)

```
>>> A = np.array([[0, 1], [1, 1], [1, 1], [2, 1]])
>>> A
array([[0, 1],
       [1, 1],
       [1, 1],
       [2, 1]])
>>> b = np.array([1, 0, 2, 1])
>>> q, r = np.linalg.qr(A)
>>> p = np.dot(q.T, b)
```

(continues on next page)

(continued from previous page)

```
>>> np.dot(np.linalg.inv(r), p)
array([  1.1e-16,   1.0e+00])
```

`numpy.linalg.`**`svd`**`(`*a*, *full_matrices=True*, *compute_uv=True*, *hermitian=False*`)`

Singular Value Decomposition.

When *a* is a 2D array, it is factorized as `u @ np.diag(s) @ vh = (u * s) @ vh`, where *u* and *vh* are 2D unitary arrays and *s* is a 1D array of *a*'s singular values. When *a* is higher-dimensional, SVD is applied in stacked mode as explained below.

**Parameters**

**a**

[(…, M, N) array_like] A real or complex array with `a.ndim >= 2`.

**full_matrices**

[bool, optional] If True (default), *u* and *vh* have the shapes `(..., M, M)` and `(..., N, N)`, respectively. Otherwise, the shapes are `(..., M, K)` and `(..., K, N)`, respectively, where `K = min(M, N)`.

**compute_uv**

[bool, optional] Whether or not to compute *u* and *vh* in addition to *s*. True by default.

**hermitian**

[bool, optional] If True, *a* is assumed to be Hermitian (symmetric if real-valued), enabling a more efficient method for finding singular values. Defaults to False.

New in version 1.17.0.

**Returns**

**u**

[{ (…, M, M), (…, M, K) } array] Unitary array(s). The first `a.ndim - 2` dimensions have the same size as those of the input *a*. The size of the last two dimensions depends on the value of *full_matrices*. Only returned when *compute_uv* is True.

**s**

[(…, K) array] Vector(s) with the singular values, within each vector sorted in descending order. The first `a.ndim - 2` dimensions have the same size as those of the input *a*.

**vh**

[{ (…, N, N), (…, K, N) } array] Unitary array(s). The first `a.ndim - 2` dimensions have the same size as those of the input *a*. The size of the last two dimensions depends on the value of *full_matrices*. Only returned when *compute_uv* is True.

**Raises**

**LinAlgError**

If SVD computation does not converge.

**See also:**

**`scipy.linalg.svd`**

Similar function in SciPy.

**`scipy.linalg.svdvals`**

Compute singular values of a matrix.

### Notes

Changed in version 1.8.0: Broadcasting rules apply, see the *`numpy.linalg`* documentation for details.

The decomposition is performed using LAPACK routine `_gesdd`.

SVD is usually described for the factorization of a 2D matrix $A$. The higher-dimensional case will be discussed below. In the 2D case, SVD is written as $A = USV^H$, where $A = a$, $U = u$, $S = \mathtt{np.diag}(s)$ and $V^H = vh$. The 1D array *s* contains the singular values of *a* and *u* and *vh* are unitary. The rows of *vh* are the eigenvectors of $A^H A$ and the columns of *u* are the eigenvectors of $AA^H$. In both cases the corresponding (possibly non-zero) eigenvalues are given by `s**2`.

If *a* has more than two dimensions, then broadcasting rules apply, as explained in *Linear algebra on several matrices at once*. This means that SVD is working in "stacked" mode: it iterates over all indices of the first `a.ndim - 2` dimensions and for each combination SVD is applied to the last two indices. The matrix *a* can be reconstructed from the decomposition with either `(u * s[..., None, :]) @ vh` or `u @ (s[..., None] * vh)`. (The `@` operator can be replaced by the function `np.matmul` for python versions below 3.5.)

If *a* is a `matrix` object (as opposed to an `ndarray`), then so are all the return values.

### Examples

```
>>> a = np.random.randn(9, 6) + 1j*np.random.randn(9, 6)
>>> b = np.random.randn(2, 7, 8, 3) + 1j*np.random.randn(2, 7, 8, 3)
```

Reconstruction based on full SVD, 2D case:

```
>>> u, s, vh = np.linalg.svd(a, full_matrices=True)
>>> u.shape, s.shape, vh.shape
((9, 9), (6,), (6, 6))
>>> np.allclose(a, np.dot(u[:, :6] * s, vh))
True
>>> smat = np.zeros((9, 6), dtype=complex)
>>> smat[:6, :6] = np.diag(s)
>>> np.allclose(a, np.dot(u, np.dot(smat, vh)))
True
```

Reconstruction based on reduced SVD, 2D case:

```
>>> u, s, vh = np.linalg.svd(a, full_matrices=False)
>>> u.shape, s.shape, vh.shape
((9, 6), (6,), (6, 6))
>>> np.allclose(a, np.dot(u * s, vh))
True
>>> smat = np.diag(s)
>>> np.allclose(a, np.dot(u, np.dot(smat, vh)))
True
```

Reconstruction based on full SVD, 4D case:

```
>>> u, s, vh = np.linalg.svd(b, full_matrices=True)
>>> u.shape, s.shape, vh.shape
((2, 7, 8, 8), (2, 7, 3), (2, 7, 3, 3))
>>> np.allclose(b, np.matmul(u[..., :3] * s[..., None, :], vh))
True
>>> np.allclose(b, np.matmul(u[..., :3], s[..., None] * vh))
True
```

Reconstruction based on reduced SVD, 4D case:

```
>>> u, s, vh = np.linalg.svd(b, full_matrices=False)
>>> u.shape, s.shape, vh.shape
((2, 7, 8, 3), (2, 7, 3), (2, 7, 3, 3))
>>> np.allclose(b, np.matmul(u * s[..., None, :], vh))
True
>>> np.allclose(b, np.matmul(u, s[..., None] * vh))
True
```

### 4.17.3 Matrix eigenvalues

| | |
|---|---|
| *linalg.eig*(a) | Compute the eigenvalues and right eigenvectors of a square array. |
| *linalg.eigh*(a[, UPLO]) | Return the eigenvalues and eigenvectors of a complex Hermitian (conjugate symmetric) or a real symmetric matrix. |
| *linalg.eigvals*(a) | Compute the eigenvalues of a general matrix. |
| *linalg.eigvalsh*(a[, UPLO]) | Compute the eigenvalues of a complex Hermitian or real symmetric matrix. |

`numpy.linalg.`**`eig`**`(`*a*`)`

Compute the eigenvalues and right eigenvectors of a square array.

**Parameters**

**a**

[(…, M, M) array] Matrices for which the eigenvalues and right eigenvectors will be computed

**Returns**

**w**

[(…, M) array] The eigenvalues, each repeated according to its multiplicity. The eigenvalues are not necessarily ordered. The resulting array will be of complex type, unless the imaginary part is zero in which case it will be cast to a real type. When *a* is real the resulting eigenvalues will be real (0 imaginary part) or occur in conjugate pairs

**v**

[(…, M, M) array] The normalized (unit "length") eigenvectors, such that the column `v[:,i]` is the eigenvector corresponding to the eigenvalue `w[i]`.

**Raises**

**LinAlgError**

If the eigenvalue computation does not converge.

**See also:**

***eigvals***

eigenvalues of a non-symmetric array.

***eigh***

eigenvalues and eigenvectors of a real symmetric or complex Hermitian (conjugate symmetric) array.

***eigvalsh***

eigenvalues of a real symmetric or complex Hermitian (conjugate symmetric) array.

**scipy.linalg.eig**

Similar function in SciPy that also solves the generalized eigenvalue problem.

**scipy.linalg.schur**

Best choice for unitary and other non-Hermitian normal matrices.

## Notes

New in version 1.8.0.

Broadcasting rules apply, see the *numpy.linalg* documentation for details.

This is implemented using the `_geev` LAPACK routines which compute the eigenvalues and eigenvectors of general square arrays.

The number *w* is an eigenvalue of *a* if there exists a vector *v* such that `a @ v = w * v`. Thus, the arrays *a*, *w*, and *v* satisfy the equations `a @ v[:,i] = w[i] * v[:,i]` for $i \in \{0, ..., M-1\}$.

The array *v* of eigenvectors may not be of maximum rank, that is, some of the columns may be linearly dependent, although round-off error may obscure that fact. If the eigenvalues are all different, then theoretically the eigenvectors are linearly independent and *a* can be diagonalized by a similarity transformation using *v*, i.e, `inv(v) @ a @ v` is diagonal.

For non-Hermitian normal matrices the SciPy function `scipy.linalg.schur` is preferred because the matrix *v* is guaranteed to be unitary, which is not the case when using *eig*. The Schur factorization produces an upper triangular matrix rather than a diagonal matrix, but for normal matrices only the diagonal of the upper triangular matrix is needed, the rest is roundoff error.

Finally, it is emphasized that *v* consists of the *right* (as in right-hand side) eigenvectors of *a*. A vector *y* satisfying `y.T @ a = z * y.T` for some number *z* is called a *left* eigenvector of *a*, and, in general, the left and right eigenvectors of a matrix are not necessarily the (perhaps conjugate) transposes of each other.

### References

G. Strang, *Linear Algebra and Its Applications*, 2nd Ed., Orlando, FL, Academic Press, Inc., 1980, Various pp.

### Examples

```
>>> from numpy import linalg as LA
```

(Almost) trivial example with real e-values and e-vectors.

```
>>> w, v = LA.eig(np.diag((1, 2, 3)))
>>> w; v
array([1., 2., 3.])
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
```

Real matrix possessing complex e-values and e-vectors; note that the e-values are complex conjugates of each other.

```
>>> w, v = LA.eig(np.array([[1, -1], [1, 1]]))
>>> w; v
array([1.+1.j, 1.-1.j])
array([[0.70710678+0.j        , 0.70710678-0.j        ],
       [0.        -0.70710678j, 0.        +0.70710678j]])
```

Complex-valued matrix with real e-values (but complex-valued e-vectors); note that `a.conj().T == a`, i.e., *a* is Hermitian.

```
>>> a = np.array([[1, 1j], [-1j, 1]])
>>> w, v = LA.eig(a)
>>> w; v
array([2.+0.j, 0.+0.j])
array([[ 0.        +0.70710678j,  0.70710678+0.j        ], # may vary
       [ 0.70710678+0.j        , -0.        +0.70710678j]])
```

Be careful about round-off error!

```
>>> a = np.array([[1 + 1e-9, 0], [0, 1 - 1e-9]])
>>> # Theor. e-values are 1 +/- 1e-9
>>> w, v = LA.eig(a)
>>> w; v
array([1., 1.])
array([[1., 0.],
       [0., 1.]])
```

`numpy.linalg.`**`eigh`**(*a*, *UPLO='L'*)

Return the eigenvalues and eigenvectors of a complex Hermitian (conjugate symmetric) or a real symmetric matrix.

Returns two objects, a 1-D array containing the eigenvalues of *a*, and a 2-D square array or matrix (depending on the input type) of the corresponding eigenvectors (in columns).

**Parameters**

**a**

[(…, M, M) array] Hermitian or real symmetric matrices whose eigenvalues and eigenvectors are to be computed.

**UPLO**

[{'L', 'U'}, optional] Specifies whether the calculation is done with the lower triangular part of *a* ('L', default) or the upper triangular part ('U'). Irrespective of this value only the real parts of the diagonal will be considered in the computation to preserve the notion of a Hermitian matrix. It therefore follows that the imaginary part of the diagonal will always be treated as zero.

**Returns**

**w**

[(…, M) ndarray] The eigenvalues in ascending order, each repeated according to its multiplicity.

**v**

[{(…, M, M) ndarray, (…, M, M) matrix}] The column `v[:, i]` is the normalized eigenvector corresponding to the eigenvalue `w[i]`. Will return a matrix object if *a* is a matrix object.

**Raises**

**LinAlgError**

If the eigenvalue computation does not converge.

**See also:**

**`eigvalsh`**

eigenvalues of real symmetric or complex Hermitian (conjugate symmetric) arrays.

**`eig`**

eigenvalues and right eigenvectors for non-symmetric arrays.

**`eigvals`**

eigenvalues of non-symmetric arrays.

**`scipy.linalg.eigh`**

Similar function in SciPy (but also solves the generalized eigenvalue problem).

### Notes

New in version 1.8.0.

Broadcasting rules apply, see the `numpy.linalg` documentation for details.

The eigenvalues/eigenvectors are computed using LAPACK routines `_syevd`, `_heevd`.

The eigenvalues of real symmetric or complex Hermitian matrices are always real. [1] The array *v* of (column) eigenvectors is unitary and *a*, *w*, and *v* satisfy the equations `dot(a, v[:, i]) = w[i] * v[:, i]`.

### References

[1]

### Examples

```
>>> from numpy import linalg as LA
>>> a = np.array([[1, -2j], [2j, 5]])
>>> a
array([[ 1.+0.j, -0.-2.j],
       [ 0.+2.j,  5.+0.j]])
>>> w, v = LA.eigh(a)
>>> w; v
array([0.17157288, 5.82842712])
array([[-0.92387953+0.j        , -0.38268343+0.j        ], # may vary
       [ 0.        +0.38268343j,  0.        -0.92387953j]])
```

```
>>> np.dot(a, v[:, 0]) - w[0] * v[:, 0] # verify 1st e-val/vec pair
array([5.55111512e-17+0.0000000e+00j, 0.00000000e+00+1.2490009e-16j])
>>> np.dot(a, v[:, 1]) - w[1] * v[:, 1] # verify 2nd e-val/vec pair
array([0.+0.j, 0.+0.j])
```

```
>>> A = np.matrix(a) # what happens if input is a matrix object
>>> A
matrix([[ 1.+0.j, -0.-2.j],
        [ 0.+2.j,  5.+0.j]])
>>> w, v = LA.eigh(A)
>>> w; v
array([0.17157288, 5.82842712])
matrix([[-0.92387953+0.j        , -0.38268343+0.j        ], # may vary
        [ 0.        +0.38268343j,  0.        -0.92387953j]])
```

```
>>> # demonstrate the treatment of the imaginary part of the diagonal
>>> a = np.array([[5+2j, 9-2j], [0+2j, 2-1j]])
>>> a
array([[5.+2.j, 9.-2.j],
       [0.+2.j, 2.-1.j]])
>>> # with UPLO='L' this is numerically equivalent to using LA.eig() with:
>>> b = np.array([[5.+0.j, 0.-2.j], [0.+2.j, 2.-0.j]])
>>> b
array([[5.+0.j, 0.-2.j],
       [0.+2.j, 2.+0.j]])
>>> wa, va = LA.eigh(a)
>>> wb, vb = LA.eig(b)
>>> wa; wb
array([1., 6.])
array([6.+0.j, 1.+0.j])
>>> va; vb
array([[-0.4472136 +0.j        , -0.89442719+0.j        ], # may vary
       [ 0.        +0.89442719j,  0.        -0.4472136j ]])
array([[ 0.89442719+0.j       , -0.        +0.4472136j],
       [-0.        +0.4472136j,  0.89442719+0.j       ]])
```

`numpy.linalg.`**`eigvals`**`(`*a*`)`

Compute the eigenvalues of a general matrix.

Main difference between `eigvals` and `eig`: the eigenvectors aren't returned.

**Parameters**

**a**

[(…, M, M) array_like] A complex- or real-valued matrix whose eigenvalues will be computed.

**Returns**

**w**

[(…, M,) ndarray] The eigenvalues, each repeated according to its multiplicity. They are not necessarily ordered, nor are they necessarily real for real matrices.

**Raises**

**LinAlgError**

If the eigenvalue computation does not converge.

**See also:**

**`eig`**

eigenvalues and right eigenvectors of general arrays

**`eigvalsh`**

eigenvalues of real symmetric or complex Hermitian (conjugate symmetric) arrays.

**`eigh`**

eigenvalues and eigenvectors of real symmetric or complex Hermitian (conjugate symmetric) arrays.

**`scipy.linalg.eigvals`**

Similar function in SciPy.

## Notes

New in version 1.8.0.

Broadcasting rules apply, see the `numpy.linalg` documentation for details.

This is implemented using the `_geev` LAPACK routines which compute the eigenvalues and eigenvectors of general square arrays.

## Examples

Illustration, using the fact that the eigenvalues of a diagonal matrix are its diagonal elements, that multiplying a matrix on the left by an orthogonal matrix, $Q$, and on the right by $Q.T$ (the transpose of $Q$), preserves the eigenvalues of the "middle" matrix. In other words, if $Q$ is orthogonal, then `Q * A * Q.T` has the same eigenvalues as `A`:

```
>>> from numpy import linalg as LA
>>> x = np.random.random()
>>> Q = np.array([[np.cos(x), -np.sin(x)], [np.sin(x), np.cos(x)]])
>>> LA.norm(Q[0, :]), LA.norm(Q[1, :]), np.dot(Q[0, :],Q[1, :])
(1.0, 1.0, 0.0)
```

Now multiply a diagonal matrix by `Q` on one side and by `Q.T` on the other:

```
>>> D = np.diag((-1,1))
>>> LA.eigvals(D)
array([-1.,  1.])
>>> A = np.dot(Q, D)
>>> A = np.dot(A, Q.T)
>>> LA.eigvals(A)
array([ 1., -1.]) # random
```

`numpy.linalg.`**`eigvalsh`**`(`*a*, *UPLO='L'*`)`

Compute the eigenvalues of a complex Hermitian or real symmetric matrix.

Main difference from eigh: the eigenvectors are not computed.

**Parameters**

**a**

[(…, M, M) array_like] A complex- or real-valued matrix whose eigenvalues are to be computed.

**UPLO**

[{'L', 'U'}, optional] Specifies whether the calculation is done with the lower triangular part of *a* ('L', default) or the upper triangular part ('U'). Irrespective of this value only the real parts of the diagonal will be considered in the computation to preserve the notion of a Hermitian matrix. It therefore follows that the imaginary part of the diagonal will always be treated as zero.

**Returns**

**w**

[(…, M,) ndarray] The eigenvalues in ascending order, each repeated according to its multiplicity.

**Raises**

**LinAlgError**

If the eigenvalue computation does not converge.

**See also:**

**`eigh`**

eigenvalues and eigenvectors of real symmetric or complex Hermitian (conjugate symmetric) arrays.

**`eigvals`**

eigenvalues of general real or complex arrays.

**`eig`**

eigenvalues and right eigenvectors of general real or complex arrays.

**`scipy.linalg.eigvalsh`**

Similar function in SciPy.

### Notes

New in version 1.8.0.

Broadcasting rules apply, see the `numpy.linalg` documentation for details.

The eigenvalues are computed using LAPACK routines `_syevd`, `_heevd`.

### Examples

```
>>> from numpy import linalg as LA
>>> a = np.array([[1, -2j], [2j, 5]])
>>> LA.eigvalsh(a)
array([ 0.17157288,  5.82842712]) # may vary
```

```
>>> # demonstrate the treatment of the imaginary part of the diagonal
>>> a = np.array([[5+2j, 9-2j], [0+2j, 2-1j]])
>>> a
array([[5.+2.j, 9.-2.j],
       [0.+2.j, 2.-1.j]])
>>> # with UPLO='L' this is numerically equivalent to using LA.eigvals()
>>> # with:
>>> b = np.array([[5.+0.j, 0.-2.j], [0.+2.j, 2.-0.j]])
>>> b
array([[5.+0.j, 0.-2.j],
       [0.+2.j, 2.+0.j]])
>>> wa = LA.eigvalsh(a)
>>> wb = LA.eigvals(b)
>>> wa; wb
array([1., 6.])
array([6.+0.j, 1.+0.j])
```

## 4.17.4 Norms and other numbers

| | |
|---|---|
| `linalg.norm`(x[, ord, axis, keepdims]) | Matrix or vector norm. |
| `linalg.cond`(x[, p]) | Compute the condition number of a matrix. |
| `linalg.det`(a) | Compute the determinant of an array. |
| `linalg.matrix_rank`(M[, tol, hermitian]) | Return matrix rank of array using SVD method |
| `linalg.slogdet`(a) | Compute the sign and (natural) logarithm of the determinant of an array. |
| `trace`(a[, offset, axis1, axis2, dtype, out]) | Return the sum along diagonals of the array. |

`numpy.linalg.`**`norm`**(*x*, *ord=None*, *axis=None*, *keepdims=False*)

Matrix or vector norm.

This function is able to return one of eight different matrix norms, or one of an infinite number of vector norms (described below), depending on the value of the `ord` parameter.

**Parameters**

**x**

[array_like] Input array. If *axis* is None, *x* must be 1-D or 2-D, unless *ord* is None. If both *axis* and *ord* are None, the 2-norm of `x.ravel` will be returned.

**ord**

[{non-zero int, inf, -inf, 'fro', 'nuc'}, optional] Order of the norm (see table under `Notes`). inf means numpy's *inf* object. The default is None.

**axis**

[{None, int, 2-tuple of ints}, optional.] If *axis* is an integer, it specifies the axis of *x* along which to compute the vector norms. If *axis* is a 2-tuple, it specifies the axes that hold 2-D matrices, and the matrix norms of these matrices are computed. If *axis* is None then either a vector norm (when *x* is 1-D) or a matrix norm (when *x* is 2-D) is returned. The default is None.

New in version 1.8.0.

**keepdims**

[bool, optional] If this is set to True, the axes which are normed over are left in the result as dimensions with size one. With this option the result will broadcast correctly against the original *x*.

New in version 1.10.0.

**Returns**

**n**

[float or ndarray] Norm of the matrix or vector(s).

**See also:**

`scipy.linalg.norm`

Similar function in SciPy.

## Notes

For values of `ord < 1`, the result is, strictly speaking, not a mathematical 'norm', but it may still be useful for various numerical purposes.

The following norms can be calculated:

| ord | norm for matrices | norm for vectors |
|---|---|---|
| None | Frobenius norm | 2-norm |
| 'fro' | Frobenius norm | – |
| 'nuc' | nuclear norm | – |
| inf | max(sum(abs(x), axis=1)) | max(abs(x)) |
| -inf | min(sum(abs(x), axis=1)) | min(abs(x)) |
| 0 | – | sum(x != 0) |
| 1 | max(sum(abs(x), axis=0)) | as below |
| -1 | min(sum(abs(x), axis=0)) | as below |
| 2 | 2-norm (largest sing. value) | as below |
| -2 | smallest singular value | as below |
| other | – | sum(abs(x)**ord)**(1./ord) |

The Frobenius norm is given by [1]:

$||A||_F = [\sum_{i,j} abs(a_{i,j})^2]^{1/2}$

The nuclear norm is the sum of the singular values.

Both the Frobenius and nuclear norm orders are only defined for matrices and raise a ValueError when `x.ndim != 2`.

### References

[1]

### Examples

```
>>> from numpy import linalg as LA
>>> a = np.arange(9) - 4
>>> a
array([-4, -3, -2, ...,  2,  3,  4])
>>> b = a.reshape((3, 3))
>>> b
array([[-4, -3, -2],
       [-1,  0,  1],
       [ 2,  3,  4]])
```

```
>>> LA.norm(a)
7.745966692414834
>>> LA.norm(b)
7.745966692414834
>>> LA.norm(b, 'fro')
7.745966692414834
>>> LA.norm(a, np.inf)
4.0
>>> LA.norm(b, np.inf)
9.0
>>> LA.norm(a, -np.inf)
0.0
>>> LA.norm(b, -np.inf)
2.0
```

```
>>> LA.norm(a, 1)
20.0
>>> LA.norm(b, 1)
7.0
>>> LA.norm(a, -1)
-4.6566128774142013e-010
>>> LA.norm(b, -1)
6.0
>>> LA.norm(a, 2)
7.745966692414834
>>> LA.norm(b, 2)
7.3484692283495345
```

```
>>> LA.norm(a, -2)
0.0
>>> LA.norm(b, -2)
1.8570331885190563e-016 # may vary
>>> LA.norm(a, 3)
```

(continues on next page)

(continued from previous page)

```
5.8480354764257312 # may vary
>>> LA.norm(a, -3)
0.0
```

Using the *axis* argument to compute vector norms:

```
>>> c = np.array([[ 1, 2, 3],
...               [-1, 1, 4]])
>>> LA.norm(c, axis=0)
array([ 1.41421356,  2.23606798,  5.        ])
>>> LA.norm(c, axis=1)
array([ 3.74165739,  4.24264069])
>>> LA.norm(c, ord=1, axis=1)
array([ 6.,  6.])
```

Using the *axis* argument to compute matrix norms:

```
>>> m = np.arange(8).reshape(2,2,2)
>>> LA.norm(m, axis=(1,2))
array([  3.74165739,  11.22497216])
>>> LA.norm(m[0, :, :]), LA.norm(m[1, :, :])
(3.7416573867739413, 11.224972160321824)
```

`numpy.linalg.`**`cond`**(*x*, *p=None*)

Compute the condition number of a matrix.

This function is capable of returning the condition number using one of seven different norms, depending on the value of *p* (see Parameters below).

**Parameters**

**x**

[(…, M, N) array_like] The matrix whose condition number is sought.

**p**

[{None, 1, -1, 2, -2, inf, -inf, 'fro'}, optional] Order of the norm:

| p | norm for matrices |
|---|---|
| None | 2-norm, computed directly using the `SVD` |
| 'fro' | Frobenius norm |
| inf | max(sum(abs(x), axis=1)) |
| -inf | min(sum(abs(x), axis=1)) |
| 1 | max(sum(abs(x), axis=0)) |
| -1 | min(sum(abs(x), axis=0)) |
| 2 | 2-norm (largest sing. value) |
| -2 | smallest singular value |

inf means the numpy.inf object, and the Frobenius norm is the root-of-sum-of-squares norm.

**Returns**

**c**

[{float, inf}] The condition number of the matrix. May be infinite.

**See also:**

*numpy.linalg.norm*

### Notes

The condition number of *x* is defined as the norm of *x* times the norm of the inverse of *x* [1]; the norm can be the usual L2-norm (root-of-sum-of-squares) or one of a number of other matrix norms.

### References

[1]

### Examples

```
>>> from numpy import linalg as LA
>>> a = np.array([[1, 0, -1], [0, 1, 0], [1, 0, 1]])
>>> a
array([[ 1,  0, -1],
       [ 0,  1,  0],
       [ 1,  0,  1]])
>>> LA.cond(a)
1.4142135623730951
>>> LA.cond(a, 'fro')
3.1622776601683795
>>> LA.cond(a, np.inf)
2.0
>>> LA.cond(a, -np.inf)
1.0
>>> LA.cond(a, 1)
2.0
>>> LA.cond(a, -1)
1.0
>>> LA.cond(a, 2)
1.4142135623730951
>>> LA.cond(a, -2)
0.70710678118654746 # may vary
>>> min(LA.svd(a, compute_uv=False))*min(LA.svd(LA.inv(a), compute_uv=False))
0.70710678118654746 # may vary
```

`numpy.linalg.`**`det`**(*a*)

Compute the determinant of an array.

**Parameters**

**a**

[(…, M, M) array_like] Input array to compute determinants for.

**Returns**

**det**

[(…) array_like] Determinant of *a*.

**See also:**

*`slogdet`*

Another way to represent the determinant, more suitable for large matrices where underflow/overflow may occur.

`scipy.linalg.det`

Similar function in SciPy.

## Notes

New in version 1.8.0.

Broadcasting rules apply, see the *`numpy.linalg`* documentation for details.

The determinant is computed via LU factorization using the LAPACK routine `z/dgetrf`.

## Examples

The determinant of a 2-D array [[a, b], [c, d]] is ad - bc:

```
>>> a = np.array([[1, 2], [3, 4]])
>>> np.linalg.det(a)
-2.0 # may vary
```

Computing determinants for a stack of matrices:

```
>>> a = np.array([ [[1, 2], [3, 4]], [[1, 2], [2, 1]], [[1, 3], [3, 1]] ])
>>> a.shape
(3, 2, 2)
>>> np.linalg.det(a)
array([-2., -3., -8.])
```

`numpy.linalg.`**`matrix_rank`**(*M*, *tol=None*, *hermitian=False*)

Return matrix rank of array using SVD method

Rank of the array is the number of singular values of the array that are greater than *tol*.

Changed in version 1.14: Can now operate on stacks of matrices

**Parameters**

**M**

[{(M,), (…, M, N)} array_like] Input vector or stack of matrices.

**tol**

[(…) array_like, float, optional] Threshold below which SVD values are considered zero. If *tol* is None, and `S` is an array with singular values for *M*, and `eps` is the epsilon value for datatype of `S`, then *tol* is set to `S.max() * max(M.shape) * eps`.

Changed in version 1.14: Broadcasted against the stack of matrices

**hermitian**

[bool, optional] If True, *M* is assumed to be Hermitian (symmetric if real-valued), enabling a more efficient method for finding singular values. Defaults to False.

New in version 1.14.

**Returns**

**rank**

[(…) array_like] Rank of M.

### Notes

The default threshold to detect rank deficiency is a test on the magnitude of the singular values of *M*. By default, we identify singular values less than `S.max() * max(M.shape) * eps` as indicating rank deficiency (with the symbols defined above). This is the algorithm MATLAB uses [1]. It also appears in *Numerical recipes* in the discussion of SVD solutions for linear least squares [2].

This default threshold is designed to detect rank deficiency accounting for the numerical errors of the SVD computation. Imagine that there is a column in *M* that is an exact (in floating point) linear combination of other columns in *M*. Computing the SVD on *M* will not produce a singular value exactly equal to 0 in general: any difference of the smallest SVD value from 0 will be caused by numerical imprecision in the calculation of the SVD. Our threshold for small SVD values takes this numerical imprecision into account, and the default threshold will detect such numerical rank deficiency. The threshold may declare a matrix *M* rank deficient even if the linear combination of some columns of *M* is not exactly equal to another column of *M* but only numerically very close to another column of *M*.

We chose our default threshold because it is in wide use. Other thresholds are possible. For example, elsewhere in the 2007 edition of *Numerical recipes* there is an alternative threshold of `S.max() * np.finfo(M.dtype).eps / 2. * np.sqrt(m + n + 1.)`. The authors describe this threshold as being based on “expected roundoff error” (p 71).

The thresholds above deal with floating point roundoff error in the calculation of the SVD. However, you may have more information about the sources of error in *M* that would make you consider other tolerance values to detect *effective* rank deficiency. The most useful measure of the tolerance depends on the operations you intend to use on your matrix. For example, if your data come from uncertain measurements with uncertainties greater than floating point epsilon, choosing a tolerance near that uncertainty may be preferable. The tolerance may be absolute if the uncertainties are absolute rather than relative.

### References

[1], [2]

### Examples

```
>>> from numpy.linalg import matrix_rank
>>> matrix_rank(np.eye(4)) # Full rank matrix
4
>>> I=np.eye(4); I[-1,-1] = 0. # rank deficient matrix
>>> matrix_rank(I)
3
>>> matrix_rank(np.ones((4,))) # 1 dimension - rank 1 unless all 0
1
>>> matrix_rank(np.zeros((4,)))
0
```

`numpy.linalg.`**`slogdet`**`(`*a*`)`

Compute the sign and (natural) logarithm of the determinant of an array.

If an array has a very small or very large determinant, then a call to `det` may overflow or underflow. This routine is more robust against such issues, because it computes the logarithm of the determinant rather than the determinant itself.

**Parameters**

**a**

[(…, M, M) array_like] Input array, has to be a square 2-D array.

**Returns**

**sign**

[(…) array_like] A number representing the sign of the determinant. For a real matrix, this is 1, 0, or -1. For a complex matrix, this is a complex number with absolute value 1 (i.e., it is on the unit circle), or else 0.

**logdet**

[(…) array_like] The natural log of the absolute value of the determinant.

**If the determinant is zero, then 'sign' will be 0 and 'logdet' will be**

**-Inf. In all cases, the determinant is equal to "sign * np.exp(logdet)".**

**See also:**

`det`

## Notes

New in version 1.8.0.

Broadcasting rules apply, see the `numpy.linalg` documentation for details.

New in version 1.6.0.

The determinant is computed via LU factorization using the LAPACK routine `z/dgetrf`.

## Examples

The determinant of a 2-D array `[[a, b], [c, d]]` is `ad - bc`:

```
>>> a = np.array([[1, 2], [3, 4]])
>>> (sign, logdet) = np.linalg.slogdet(a)
>>> (sign, logdet)
(-1, 0.69314718055994529) # may vary
>>> sign * np.exp(logdet)
-2.0
```

Computing log-determinants for a stack of matrices:

```
>>> a = np.array([ [[1, 2], [3, 4]], [[1, 2], [2, 1]], [[1, 3], [3, 1]] ])
>>> a.shape
(3, 2, 2)
>>> sign, logdet = np.linalg.slogdet(a)
>>> (sign, logdet)
(array([-1., -1., -1.]), array([ 0.69314718,  1.09861229,  2.07944154]))
>>> sign * np.exp(logdet)
array([-2., -3., -8.])
```

This routine succeeds where ordinary `det` does not:

```
>>> np.linalg.det(np.eye(500) * 0.1)
0.0
>>> np.linalg.slogdet(np.eye(500) * 0.1)
(1, -1151.2925464970228)
```

numpy.**trace**(*a*, *offset=0*, *axis1=0*, *axis2=1*, *dtype=None*, *out=None*)

Return the sum along diagonals of the array.

If *a* is 2-D, the sum along its diagonal with the given offset is returned, i.e., the sum of elements `a[i,i+offset]` for all i.

If *a* has more than two dimensions, then the axes specified by axis1 and axis2 are used to determine the 2-D sub-arrays whose traces are returned. The shape of the resulting array is the same as that of *a* with *axis1* and *axis2* removed.

**Parameters**

**a**

[array_like] Input array, from which the diagonals are taken.

**offset**

[int, optional] Offset of the diagonal from the main diagonal. Can be both positive and negative. Defaults to 0.

**axis1, axis2**

[int, optional] Axes to be used as the first and second axis of the 2-D sub-arrays from which the diagonals should be taken. Defaults are the first two axes of *a*.

**dtype**

[dtype, optional] Determines the data-type of the returned array and of the accumulator where the elements are summed. If dtype has the value None and *a* is of integer type of precision less than the default integer precision, then the default integer precision is used. Otherwise, the precision is the same as that of *a*.

**out**

[ndarray, optional] Array into which the output is placed. Its type is preserved and it must be of the right shape to hold the output.

**Returns**

**sum_along_diagonals**

[ndarray] If *a* is 2-D, the sum along the diagonal is returned. If *a* has larger dimensions, then an array of sums along diagonals is returned.

**See also:**

*diag*, *diagonal*, *diagflat*

### Examples

```
>>> np.trace(np.eye(3))
3.0
>>> a = np.arange(8).reshape((2,2,2))
>>> np.trace(a)
array([6, 8])
```

```
>>> a = np.arange(24).reshape((2,2,2,3))
>>> np.trace(a).shape
(2, 3)
```

## 4.17.5 Solving equations and inverting matrices

| | |
|---|---|
| *linalg.solve*(a, b) | Solve a linear matrix equation, or system of linear scalar equations. |
| *linalg.tensorsolve*(a, b[, axes]) | Solve the tensor equation `a x = b` for x. |
| *linalg.lstsq*(a, b[, rcond]) | Return the least-squares solution to a linear matrix equation. |
| *linalg.inv*(a) | Compute the (multiplicative) inverse of a matrix. |
| *linalg.pinv*(a[, rcond, hermitian]) | Compute the (Moore-Penrose) pseudo-inverse of a matrix. |
| *linalg.tensorinv*(a[, ind]) | Compute the 'inverse' of an N-dimensional array. |

`numpy.linalg.`**`solve`**`(`*a*, *b*`)`

Solve a linear matrix equation, or system of linear scalar equations.

Computes the "exact" solution, *x*, of the well-determined, i.e., full rank, linear matrix equation $ax = b$.

**Parameters**

**a**

[(…, M, M) array_like] Coefficient matrix.

**b**

[{(…, M,), (…, M, K)}, array_like] Ordinate or "dependent variable" values.

**Returns**

**x**

[{(…, M,), (…, M, K)} ndarray] Solution to the system a x = b. Returned shape is identical to *b*.

**Raises**

**LinAlgError**

If *a* is singular or not square.

**See also:**

**`scipy.linalg.solve`**
Similar function in SciPy.

## Notes

New in version 1.8.0.

Broadcasting rules apply, see the *`numpy.linalg`* documentation for details.

The solutions are computed using LAPACK routine `_gesv`.

*a* must be square and of full-rank, i.e., all rows (or, equivalently, columns) must be linearly independent; if either is not true, use *`lstsq`* for the least-squares best "solution" of the system/equation.

## References

[1]

## Examples

Solve the system of equations `3 * x0 + x1 = 9` and `x0 + 2 * x1 = 8`:

```
>>> a = np.array([[3,1], [1,2]])
>>> b = np.array([9,8])
>>> x = np.linalg.solve(a, b)
>>> x
array([2.,  3.])
```

Check that the solution is correct:

```
>>> np.allclose(np.dot(a, x), b)
True
```

`numpy.linalg.`**`tensorsolve`**(*a*, *b*, *axes=None*)
Solve the tensor equation `a x = b` for x.

It is assumed that all indices of *x* are summed over in the product, together with the rightmost indices of *a*, as is done in, for example, `tensordot(a, x, axes=b.ndim)`.

**Parameters**

**a**

[array_like] Coefficient tensor, of shape `b.shape + Q`. *Q*, a tuple, equals the shape of that sub-tensor of *a* consisting of the appropriate number of its rightmost indices, and must be such that `prod(Q) == prod(b.shape)` (in which sense *a* is said to be 'square').

**b**

[array_like] Right-hand tensor, which can be of any shape.

**axes**

[tuple of ints, optional] Axes in *a* to reorder to the right, before inversion. If None (default), no reordering is done.

**Returns**

**x**

[ndarray, shape Q]

**Raises**

**LinAlgError**

If *a* is singular or not 'square' (in the above sense).

**See also:**

*numpy.tensordot*, *tensorinv*, *numpy.einsum*

## Examples

```
>>> a = np.eye(2*3*4)
>>> a.shape = (2*3, 4, 2, 3, 4)
>>> b = np.random.randn(2*3, 4)
>>> x = np.linalg.tensorsolve(a, b)
>>> x.shape
(2, 3, 4)
>>> np.allclose(np.tensordot(a, x, axes=3), b)
True
```

`numpy.linalg.`**`lstsq`**(*a*, *b*, *rcond='warn'*)

Return the least-squares solution to a linear matrix equation.

Computes the vector x that approximatively solves the equation `a @ x = b`. The equation may be under-, well-, or over-determined (i.e., the number of linearly independent rows of *a* can be less than, equal to, or greater than its number of linearly independent columns). If *a* is square and of full rank, then *x* (but for round-off error) is the "exact" solution of the equation. Else, *x* minimizes the Euclidean 2-norm $||b - ax||$.

**Parameters**

**a**

[(M, N) array_like] "Coefficient" matrix.

**b**

[{(M,), (M, K)} array_like] Ordinate or "dependent variable" values. If *b* is two-dimensional, the least-squares solution is calculated for each of the *K* columns of *b*.

**rcond**

[float, optional] Cut-off ratio for small singular values of *a*. For the purposes of rank determination, singular values are treated as zero if they are smaller than *rcond* times the largest singular value of *a*.

Changed in version 1.14.0: If not set, a FutureWarning is given. The previous default of `-1` will use the machine precision as *rcond* parameter, the new default will use the machine precision times *max(M, N)*. To silence the warning and use the new default, use `rcond=None`, to keep using the old behavior, use `rcond=-1`.

**Returns**

**x**

[{(N,), (N, K)} ndarray] Least-squares solution. If *b* is two-dimensional, the solutions are in the *K* columns of *x*.

**residuals**

[{(1,), (K,), (0,)} ndarray] Sums of residuals; squared Euclidean 2-norm for each column in `b - a*x`. If the rank of *a* is < N or M <= N, this is an empty array. If *b* is 1-dimensional, this is a (1,) shape array. Otherwise the shape is (K,).

**rank**

[int] Rank of matrix *a*.

**s**

[(min(M, N),) ndarray] Singular values of *a*.

**Raises**

**LinAlgError**

If computation does not converge.

**See also:**

**`scipy.linalg.lstsq`**

Similar function in SciPy.

## Notes

If *b* is a matrix, then all array results are returned as matrices.

## Examples

Fit a line, `y = mx + c`, through some noisy data-points:

```
>>> x = np.array([0, 1, 2, 3])
>>> y = np.array([-1, 0.2, 0.9, 2.1])
```

By examining the coefficients, we see that the line should have a gradient of roughly 1 and cut the y-axis at, more or less, -1.

We can rewrite the line equation as `y = Ap`, where `A = [[x 1]]` and `p = [[m], [c]]`. Now use *`lstsq`* to solve for *p*:

```
>>> A = np.vstack([x, np.ones(len(x))]).T
>>> A
array([[ 0.,  1.],
       [ 1.,  1.],
```

(continues on next page)

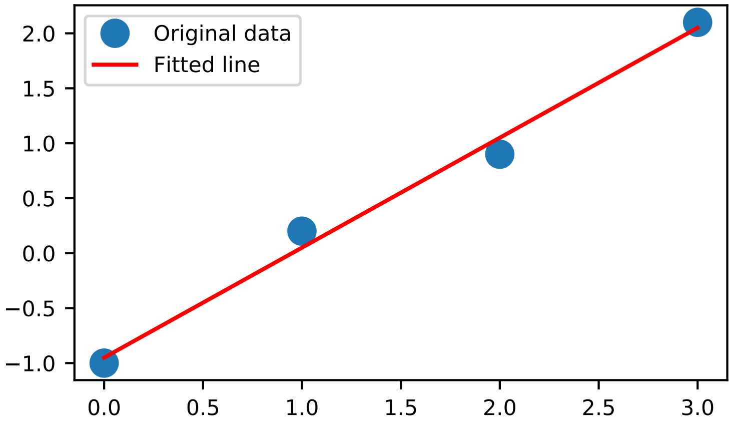
(continued from previous page)

```
        [ 2.,  1.],
        [ 3.,  1.]])
```

```
>>> m, c = np.linalg.lstsq(A, y, rcond=None)[0]
>>> m, c
(1.0 -0.95) # may vary
```

Plot the data along with the fitted line:

```
>>> import matplotlib.pyplot as plt
>>> _ = plt.plot(x, y, 'o', label='Original data', markersize=10)
>>> _ = plt.plot(x, m*x + c, 'r', label='Fitted line')
>>> _ = plt.legend()
>>> plt.show()
```



`numpy.linalg.`**`inv`**`(`*a*`)`

Compute the (multiplicative) inverse of a matrix.

Given a square matrix *a*, return the matrix *ainv* satisfying `dot(a, ainv) = dot(ainv, a) = eye(a.shape[0])`.

**Parameters**

**a**

[(…, M, M) array_like] Matrix to be inverted.

**Returns**

**ainv**

[(…, M, M) ndarray or matrix] (Multiplicative) inverse of the matrix *a*.

**Raises**

**LinAlgError**

If *a* is not square or inversion fails.

**See also:**

**`scipy.linalg.inv`**
Similar function in SciPy.

## Notes

New in version 1.8.0.

Broadcasting rules apply, see the *`numpy.linalg`* documentation for details.

## Examples

```
>>> from numpy.linalg import inv
>>> a = np.array([[1., 2.], [3., 4.]])
>>> ainv = inv(a)
>>> np.allclose(np.dot(a, ainv), np.eye(2))
True
>>> np.allclose(np.dot(ainv, a), np.eye(2))
True
```

If a is a matrix object, then the return value is a matrix as well:

```
>>> ainv = inv(np.matrix(a))
>>> ainv
matrix([[-2. ,  1. ],
        [ 1.5, -0.5]])
```

Inverses of several matrices can be computed at once:

```
>>> a = np.array([[[1., 2.], [3., 4.]], [[1, 3], [3, 5]]])
>>> inv(a)
array([[[-2.  ,  1.  ],
        [ 1.5 , -0.5 ]],
       [[-1.25,  0.75],
        [ 0.75, -0.25]]])
```

`numpy.linalg.`**`pinv`**(*a*, *rcond=1e-15*, *hermitian=False*)

Compute the (Moore-Penrose) pseudo-inverse of a matrix.

Calculate the generalized inverse of a matrix using its singular-value decomposition (SVD) and including all *large* singular values.

Changed in version 1.14: Can now operate on stacks of matrices

**Parameters**

**a**

[(…, M, N) array_like] Matrix or stack of matrices to be pseudo-inverted.

**rcond**

[(…) array_like of float] Cutoff for small singular values. Singular values less than or equal to `rcond * largest_singular_value` are set to zero. Broadcasts against the stack of matrices.

**hermitian**

[bool, optional] If True, *a* is assumed to be Hermitian (symmetric if real-valued), enabling a more efficient method for finding singular values. Defaults to False.

New in version 1.17.0.

**Returns**

**B**

[(…, N, M) ndarray] The pseudo-inverse of *a*. If *a* is a *matrix* instance, then so is *B*.

**Raises**

**LinAlgError**

If the SVD computation does not converge.

**See also:**

**`scipy.linalg.pinv`**

Similar function in SciPy.

**`scipy.linalg.pinv2`**

Similar function in SciPy (SVD-based).

**`scipy.linalg.pinvh`**

Compute the (Moore-Penrose) pseudo-inverse of a Hermitian matrix.

## Notes

The pseudo-inverse of a matrix A, denoted $A^+$, is defined as: "the matrix that 'solves' [the least-squares problem] $Ax = b$," i.e., if $\bar{x}$ is said solution, then $A^+$ is that matrix such that $\bar{x} = A^+b$.

It can be shown that if $Q_1 \Sigma Q_2^T = A$ is the singular value decomposition of A, then $A^+ = Q_2 \Sigma^+ Q_1^T$, where $Q_{1,2}$ are orthogonal matrices, $\Sigma$ is a diagonal matrix consisting of A's so-called singular values, (followed, typically, by zeros), and then $\Sigma^+$ is simply the diagonal matrix consisting of the reciprocals of A's singular values (again, followed by zeros). [1]

## References

[1]

## Examples

The following example checks that `a * a+ * a == a` and `a+ * a * a+ == a+`:

```
>>> a = np.random.randn(9, 6)
>>> B = np.linalg.pinv(a)
>>> np.allclose(a, np.dot(a, np.dot(B, a)))
True
>>> np.allclose(B, np.dot(B, np.dot(a, B)))
True
```

`numpy.linalg.`**`tensorinv`**`(`*a*, *ind=2*`)`

Compute the 'inverse' of an N-dimensional array.

The result is an inverse for *a* relative to the tensordot operation `tensordot(a, b, ind)`, i. e., up to floating-point accuracy, `tensordot(tensorinv(a), a, ind)` is the "identity" tensor for the tensordot operation.

**Parameters**

**a**

[array_like] Tensor to 'invert'. Its shape must be 'square', i. e., `prod(a.shape[:ind]) == prod(a.shape[ind:])`.

**ind**

[int, optional] Number of first indices that are involved in the inverse sum. Must be a positive integer, default is 2.

**Returns**

**b**

[ndarray] *a*'s tensordot inverse, shape `a.shape[ind:] + a.shape[:ind]`.

**Raises**

**LinAlgError**

If *a* is singular or not 'square' (in the above sense).

**See also:**

*`numpy.tensordot`*, *`tensorsolve`*

## Examples

```
>>> a = np.eye(4*6)
>>> a.shape = (4, 6, 8, 3)
>>> ainv = np.linalg.tensorinv(a, ind=2)
>>> ainv.shape
(8, 3, 4, 6)
>>> b = np.random.randn(4, 6)
>>> np.allclose(np.tensordot(ainv, b), np.linalg.tensorsolve(a, b))
True
```

```
>>> a = np.eye(4*6)
>>> a.shape = (24, 8, 3)
>>> ainv = np.linalg.tensorinv(a, ind=1)
>>> ainv.shape
(8, 3, 24)
>>> b = np.random.randn(24)
>>> np.allclose(np.tensordot(ainv, b, 1), np.linalg.tensorsolve(a, b))
True
```

### 4.17.6 Exceptions

| | |
|---|---|
| *linalg.LinAlgError* | Generic Python-exception-derived object raised by linalg functions. |

**exception** `numpy.linalg.`**`LinAlgError`**

Generic Python-exception-derived object raised by linalg functions.

General purpose exception class, derived from Python's exception.Exception class, programmatically raised in linalg functions when a Linear Algebra-related condition would prevent further correct execution of the function.

**Parameters**

**None**

#### Examples

```
>>> from numpy import linalg as LA
>>> LA.inv(np.zeros((2,2)))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "...linalg.py", line 350,
    in inv return wrap(solve(a, identity(a.shape[0], dtype=a.dtype)))
  File "...linalg.py", line 249,
    in solve
    raise LinAlgError('Singular matrix')
numpy.linalg.LinAlgError: Singular matrix
```

### 4.17.7 Linear algebra on several matrices at once

New in version 1.8.0.

Several of the linear algebra routines listed above are able to compute results for several matrices at once, if they are stacked into the same array.

This is indicated in the documentation via input parameter specifications such as `a : (..., M, M) array_like`. This means that if for instance given an input array `a.shape == (N, M, M)`, it is interpreted as a "stack" of N matrices, each of size M-by-M. Similar specification applies to return values, for instance the determinant has `det : (...)` and will in this case return an array of shape `det(a).shape == (N,)`. This generalizes to linear algebra operations on higher-dimensional arrays: the last 1 or 2 dimensions of a multidimensional array are interpreted as vectors or matrices, as appropriate for each operation.

# 4.18 Logic functions

## 4.18.1 Truth value testing

| | |
|---|---|
| `all`(a[, axis, out, keepdims]) | Test whether all array elements along a given axis evaluate to True. |
| `any`(a[, axis, out, keepdims]) | Test whether any array element along a given axis evaluates to True. |

`numpy.`**`all`**`(`*a*, *axis=None*, *out=None*, *keepdims=<no value>*`)`

Test whether all array elements along a given axis evaluate to True.

**Parameters**

**a**

[array_like] Input array or object that can be converted to an array.

**axis**

[None or int or tuple of ints, optional] Axis or axes along which a logical AND reduction is performed. The default (`axis=None`) is to perform a logical AND over all the dimensions of the input array. *axis* may be negative, in which case it counts from the last to the first axis.

New in version 1.7.0.

If this is a tuple of ints, a reduction is performed on multiple axes, instead of a single axis or all the axes as before.

**out**

[ndarray, optional] Alternate output array in which to place the result. It must have the same shape as the expected output and its type is preserved (e.g., if `dtype(out)` is float, the result will consist of 0.0's and 1.0's). See *ufuncs-output-type* for more details.

**keepdims**

[bool, optional] If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the input array.

If the default value is passed, then *keepdims* will not be passed through to the `all` method of sub-classes of `ndarray`, however any non-default value will be. If the sub-class' method does not implement *keepdims* any exceptions will be raised.

**Returns**

**all**

[ndarray, bool] A new boolean or array is returned unless *out* is specified, in which case a reference to *out* is returned.

**See also:**

**`ndarray.all`**

equivalent method

*`any`*

Test whether any element along a given axis evaluates to True.

### Notes

Not a Number (NaN), positive infinity and negative infinity evaluate to *True* because these are not equal to zero.

### Examples

```
>>> np.all([[True,False],[True,True]])
False
```

```
>>> np.all([[True,False],[True,True]], axis=0)
array([ True, False])
```

```
>>> np.all([-1, 4, 5])
True
```

```
>>> np.all([1.0, np.nan])
True
```

```
>>> o=np.array(False)
>>> z=np.all([-1, 4, 5], out=o)
>>> id(z), id(o), z
(28293632, 28293632, array(True)) # may vary
```

`numpy.`**`any`**(*a*, *axis=None*, *out=None*, *keepdims=<no value>*)

Test whether any array element along a given axis evaluates to True.

Returns single boolean unless *axis* is not `None`

**Parameters**

**a**

[array_like] Input array or object that can be converted to an array.

**axis**

[None or int or tuple of ints, optional] Axis or axes along which a logical OR reduction is performed. The default (`axis=None`) is to perform a logical OR over all the dimensions of the input array. *axis* may be negative, in which case it counts from the last to the first axis.

New in version 1.7.0.

If this is a tuple of ints, a reduction is performed on multiple axes, instead of a single axis or all the axes as before.

**out**

[ndarray, optional] Alternate output array in which to place the result. It must have the same shape as the expected output and its type is preserved (e.g., if it is of type float, then it will remain so, returning 1.0 for True and 0.0 for False, regardless of the type of *a*). See *ufuncs-output-type* for more details.

**keepdims**

[bool, optional] If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the input array.

If the default value is passed, then *keepdims* will not be passed through to the `any` method of sub-classes of `ndarray`, however any non-default value will be. If the sub-class' method does not implement *keepdims* any exceptions will be raised.

**Returns**

**any**

[bool or ndarray] A new boolean or `ndarray` is returned unless *out* is specified, in which case a reference to *out* is returned.

**See also:**

***`ndarray.any`***

equivalent method

***`all`***

Test whether all elements along a given axis evaluate to True.

## Notes

Not a Number (NaN), positive infinity and negative infinity evaluate to *True* because these are not equal to zero.

## Examples

```
>>> np.any([[True, False], [True, True]])
True
```

```
>>> np.any([[True, False], [False, False]], axis=0)
array([ True, False])
```

```
>>> np.any([-1, 0, 5])
True
```

```
>>> np.any(np.nan)
True
```

```
>>> o=np.array(False)
>>> z=np.any([-1, 4, 5], out=o)
>>> z, o
(array(True), array(True))
>>> # Check now that z is a reference to o
>>> z is o
True
>>> id(z), id(o) # identity of z and o
(191614240, 191614240)
```

## 4.18.2 Array contents

| | |
|---|---|
| `isfinite`(x, /[, out, where, casting, order, …]) | Test element-wise for finiteness (not infinity or not Not a Number). |
| `isinf`(x, /[, out, where, casting, order, …]) | Test element-wise for positive or negative infinity. |
| `isnan`(x, /[, out, where, casting, order, …]) | Test element-wise for NaN and return result as a boolean array. |
| `isnat`(x, /[, out, where, casting, order, …]) | Test element-wise for NaT (not a time) and return result as a boolean array. |
| `isneginf`(x[, out]) | Test element-wise for negative infinity, return result as bool array. |
| `isposinf`(x[, out]) | Test element-wise for positive infinity, return result as bool array. |

`numpy.`**`isfinite`**(*x*, */*, *out=None*, *, *where=True*, *casting='same_kind'*, *order='K'*, *dtype=None*, *subok=True*[, *signature*, *extobj*]) **`= <ufunc 'isfinite'>`**

Test element-wise for finiteness (not infinity or not Not a Number).

The result is returned as a boolean array.

**Parameters**

**x**

[array_like] Input values.

**out**

[ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

**where**

[array_like, optional] This condition is broadcast over the input. At locations where the condition is True, the *out* array will be set to the ufunc result. Elsewhere, the *out* array will retain its original value. Note that if an uninitialized *out* array is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.

****kwargs**

For other keyword-only arguments, see the *ufunc docs*.

**Returns**

**y**

[ndarray, bool] True where `x` is not positive infinity, negative infinity, or NaN; false otherwise. This is a scalar if *x* is a scalar.

**See also:**

`isinf`, `isneginf`, `isposinf`, `isnan`

## Notes

Not a Number, positive infinity and negative infinity are considered to be non-finite.

NumPy uses the IEEE Standard for Binary Floating-Point for Arithmetic (IEEE 754). This means that Not a Number is not equivalent to infinity. Also that positive infinity is not equivalent to negative infinity. But infinity is equivalent to positive infinity. Errors result if the second argument is also supplied when *x* is a scalar input, or if first and second arguments have different shapes.

## Examples

```
>>> np.isfinite(1)
True
>>> np.isfinite(0)
True
>>> np.isfinite(np.nan)
False
>>> np.isfinite(np.inf)
False
>>> np.isfinite(np.NINF)
False
>>> np.isfinite([np.log(-1.),1.,np.log(0)])
array([False,  True, False])
```

```
>>> x = np.array([-np.inf, 0., np.inf])
>>> y = np.array([2, 2, 2])
>>> np.isfinite(x, y)
array([0, 1, 0])
>>> y
array([0, 1, 0])
```

`numpy.`**`isinf`**(*x*, */*, *out=None*, ***, *where=True*, *casting='same_kind'*, *order='K'*, *dtype=None*, *subok=True*[, *signature*, *extobj* ]) **`= <ufunc 'isinf'>`**

Test element-wise for positive or negative infinity.

Returns a boolean array of the same shape as *x*, True where `x == +/-inf`, otherwise False.

**Parameters**

**x**

[array_like] Input values

**out**

[ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

**where**

[array_like, optional] This condition is broadcast over the input. At locations where the condition is True, the *out* array will be set to the ufunc result. Elsewhere, the *out* array will retain its original value. Note that if an uninitialized *out* array is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.

****kwargs**

For other keyword-only arguments, see the *ufunc docs*.

**Returns**

**y**

[bool (scalar) or boolean ndarray] True where `x` is positive or negative infinity, false otherwise. This is a scalar if *x* is a scalar.

**See also:**

`isneginf`, `isposinf`, `isnan`, `isfinite`

## Notes

NumPy uses the IEEE Standard for Binary Floating-Point for Arithmetic (IEEE 754).

Errors result if the second argument is supplied when the first argument is a scalar, or if the first and second arguments have different shapes.

## Examples

```
>>> np.isinf(np.inf)
True
>>> np.isinf(np.nan)
False
>>> np.isinf(np.NINF)
True
>>> np.isinf([np.inf, -np.inf, 1.0, np.nan])
array([ True,  True, False, False])
```

```
>>> x = np.array([-np.inf, 0., np.inf])
>>> y = np.array([2, 2, 2])
>>> np.isinf(x, y)
array([1, 0, 1])
>>> y
array([1, 0, 1])
```

`numpy.`**`isnan`**(*x*, */*, *out=None*, ***, *where=True*, *casting='same_kind'*, *order='K'*, *dtype=None*, *subok=True*[, *signature*, *extobj* ]) **`= <ufunc 'isnan'>`**

Test element-wise for NaN and return result as a boolean array.

**Parameters**

**x**

[array_like] Input array.

**out**

[ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

**where**

[array_like, optional] This condition is broadcast over the input. At locations where the condition is True, the *out* array will be set to the ufunc result. Elsewhere, the *out* array will retain its original value. Note that if an uninitialized *out* array is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.

****kwargs**

For other keyword-only arguments, see the *ufunc docs*.

**Returns**

**y**

[ndarray or bool] True where `x` is NaN, false otherwise. This is a scalar if *x* is a scalar.

**See also:**

`isinf`, `isneginf`, `isposinf`, `isfinite`, `isnat`

### Notes

NumPy uses the IEEE Standard for Binary Floating-Point for Arithmetic (IEEE 754). This means that Not a Number is not equivalent to infinity.

### Examples

```
>>> np.isnan(np.nan)
True
>>> np.isnan(np.inf)
False
>>> np.isnan([np.log(-1.),1.,np.log(0)])
array([ True, False, False])
```

`numpy.`**`isnat`** (*x*, */*, *out=None*, ***, *where=True*, *casting='same_kind'*, *order='K'*, *dtype=None*, *subok=True*[, *signature*, *extobj*]) **`= <ufunc 'isnat'>`**

Test element-wise for NaT (not a time) and return result as a boolean array.

New in version 1.13.0.

**Parameters**

**x**

[array_like] Input array with datetime or timedelta data type.

**out**

[ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

**where**

[array_like, optional] This condition is broadcast over the input. At locations where the condition is True, the *out* array will be set to the ufunc result. Elsewhere, the *out* array will retain

its original value. Note that if an uninitialized *out* array is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.

****kwargs**

For other keyword-only arguments, see the *ufunc docs*.

**Returns**

**y**

[ndarray or bool] True where `x` is NaT, false otherwise. This is a scalar if *x* is a scalar.

**See also:**

`isnan`, `isinf`, `isneginf`, `isposinf`, `isfinite`

## Examples

```
>>> np.isnat(np.datetime64("NaT"))
True
>>> np.isnat(np.datetime64("2016-01-01"))
False
>>> np.isnat(np.array(["NaT", "2016-01-01"], dtype="datetime64[ns]"))
array([ True, False])
```

`numpy.`**`isneginf`**(*x*, *out=None*)

Test element-wise for negative infinity, return result as bool array.

**Parameters**

**x**

[array_like] The input array.

**out**

[array_like, optional] A location into which the result is stored. If provided, it must have a shape that the input broadcasts to. If not provided or None, a freshly-allocated boolean array is returned.

**Returns**

**out**

[ndarray] A boolean array with the same dimensions as the input. If second argument is not supplied then a numpy boolean array is returned with values True where the corresponding element of the input is negative infinity and values False where the element of the input is not negative infinity.

If a second argument is supplied the result is stored there. If the type of that array is a numeric type the result is represented as zeros and ones, if the type is boolean then as False and True. The return value *out* is then a reference to that array.

**See also:**

`isinf`, `isposinf`, `isnan`, `isfinite`

## Notes

NumPy uses the IEEE Standard for Binary Floating-Point for Arithmetic (IEEE 754).

Errors result if the second argument is also supplied when x is a scalar input, if first and second arguments have different shapes, or if the first argument has complex values.

## Examples

```
>>> np.isneginf(np.NINF)
True
>>> np.isneginf(np.inf)
False
>>> np.isneginf(np.PINF)
False
>>> np.isneginf([-np.inf, 0., np.inf])
array([ True, False, False])
```

```
>>> x = np.array([-np.inf, 0., np.inf])
>>> y = np.array([2, 2, 2])
>>> np.isneginf(x, y)
array([1, 0, 0])
>>> y
array([1, 0, 0])
```

`numpy.`**`isposinf`**(*x*, *out=None*)

Test element-wise for positive infinity, return result as bool array.

**Parameters**

**x**

[array_like] The input array.

**out**

[array_like, optional] A location into which the result is stored. If provided, it must have a shape that the input broadcasts to. If not provided or None, a freshly-allocated boolean array is returned.

**Returns**

**out**

[ndarray] A boolean array with the same dimensions as the input. If second argument is not supplied then a boolean array is returned with values True where the corresponding element of the input is positive infinity and values False where the element of the input is not positive infinity.

If a second argument is supplied the result is stored there. If the type of that array is a numeric type the result is represented as zeros and ones, if the type is boolean then as False and True. The return value *out* is then a reference to that array.

**See also:**

*`isinf`*, *`isneginf`*, *`isfinite`*, *`isnan`*

### Notes

NumPy uses the IEEE Standard for Binary Floating-Point for Arithmetic (IEEE 754).

Errors result if the second argument is also supplied when x is a scalar input, if first and second arguments have different shapes, or if the first argument has complex values

### Examples

```
>>> np.isposinf(np.PINF)
True
>>> np.isposinf(np.inf)
True
>>> np.isposinf(np.NINF)
False
>>> np.isposinf([-np.inf, 0., np.inf])
array([False, False,  True])
```

```
>>> x = np.array([-np.inf, 0., np.inf])
>>> y = np.array([2, 2, 2])
>>> np.isposinf(x, y)
array([0, 0, 1])
>>> y
array([0, 0, 1])
```

## 4.18.3 Array type testing

| | |
|---|---|
| `iscomplex`(x) | Returns a bool array, where True if input element is complex. |
| `iscomplexobj`(x) | Check for a complex type or an array of complex numbers. |
| `isfortran`(a) | Check if the array is Fortran contiguous but *not* C contiguous. |
| `isreal`(x) | Returns a bool array, where True if input element is real. |
| `isrealobj`(x) | Return True if x is a not complex type or an array of complex numbers. |
| `isscalar`(element) | Returns True if the type of *element* is a scalar type. |

`numpy.`**`iscomplex`**(*x*)

Returns a bool array, where True if input element is complex.

What is tested is whether the input has a non-zero imaginary part, not if the input type is complex.

**Parameters**

**x**

[array_like] Input array.

**Returns**

**out**

[ndarray of bools] Output array.

**See also:**

*isreal*

***iscomplexobj***

Return True if x is a complex type or an array of complex numbers.

### Examples

```
>>> np.iscomplex([1+1j, 1+0j, 4.5, 3, 2, 2j])
array([ True, False, False, False, False,  True])
```

`numpy.`**`iscomplexobj`**`(x)`

Check for a complex type or an array of complex numbers.

The type of the input is checked, not the value. Even if the input has an imaginary part equal to zero, *iscomplexobj* evaluates to True.

**Parameters**

**x**

[any] The input can be of any type and shape.

**Returns**

**iscomplexobj**

[bool] The return value, True if *x* is of a complex type or has at least one complex element.

**See also:**

*isrealobj*, *iscomplex*

### Examples

```
>>> np.iscomplexobj(1)
False
>>> np.iscomplexobj(1+0j)
True
>>> np.iscomplexobj([3, 1+0j, True])
True
```

`numpy.`**`isfortran`**`(a)`

Check if the array is Fortran contiguous but *not* C contiguous.

This function is obsolete and, because of changes due to relaxed stride checking, its return value for the same array may differ for versions of NumPy >= 1.10.0 and previous versions. If you only want to check if an array is Fortran contiguous use `a.flags.f_contiguous` instead.

**Parameters**

**a**

[ndarray] Input array.

**Returns**

**isfortran**

[bool] Returns True if the array is Fortran contiguous but *not* C contiguous.

### Examples

np.array allows to specify whether the array is written in C-contiguous order (last index varies the fastest), or FORTRAN-contiguous order in memory (first index varies the fastest).

```
>>> a = np.array([[1, 2, 3], [4, 5, 6]], order='C')
>>> a
array([[1, 2, 3],
       [4, 5, 6]])
>>> np.isfortran(a)
False
```

```
>>> b = np.array([[1, 2, 3], [4, 5, 6]], order='F')
>>> b
array([[1, 2, 3],
       [4, 5, 6]])
>>> np.isfortran(b)
True
```

The transpose of a C-ordered array is a FORTRAN-ordered array.

```
>>> a = np.array([[1, 2, 3], [4, 5, 6]], order='C')
>>> a
array([[1, 2, 3],
       [4, 5, 6]])
>>> np.isfortran(a)
False
>>> b = a.T
>>> b
array([[1, 4],
       [2, 5],
       [3, 6]])
>>> np.isfortran(b)
True
```

C-ordered arrays evaluate as False even if they are also FORTRAN-ordered.

```
>>> np.isfortran(np.array([1, 2], order='F'))
False
```

`numpy.`**`isreal`**`(`*x*`)`

Returns a bool array, where True if input element is real.

If element has complex type with zero complex part, the return value for that element is True.

**Parameters**

**x**

[array_like] Input array.

**Returns**

**out**

[ndarray, bool] Boolean array of same shape as *x*.

**See also:**

*iscomplex*

***isrealobj***

Return True if x is not a complex type.

## Examples

```
>>> np.isreal([1+1j, 1+0j, 4.5, 3, 2, 2j])
array([False,  True,  True,  True,  True, False])
```

`numpy.`**`isrealobj`**`(`*x*`)`

Return True if x is a not complex type or an array of complex numbers.

The type of the input is checked, not the value. So even if the input has an imaginary part equal to zero, *isrealobj* evaluates to False if the data type is complex.

**Parameters**

**x**

[any] The input can be of any type and shape.

**Returns**

**y**

[bool] The return value, False if *x* is of a complex type.

**See also:**

*iscomplexobj*, *isreal*

## Examples

```
>>> np.isrealobj(1)
True
>>> np.isrealobj(1+0j)
False
>>> np.isrealobj([3, 1+0j, True])
False
```

`numpy.`**`isscalar`**`(`*element*`)`

Returns True if the type of *element* is a scalar type.

**Parameters**

**element**

[any] Input argument, can be of any type and shape.

**Returns**

**val**

[bool] True if *element* is a scalar type, False if it is not.

**See also:**

**`ndim`**

Get the number of dimensions of an array

## Notes

If you need a stricter way to identify a *numerical* scalar, use `isinstance(x, numbers.Number)`, as that returns `False` for most non-numerical elements such as strings.

In most cases `np.ndim(x) == 0` should be used instead of this function, as that will also return true for 0d arrays. This is how numpy overloads functions in the style of the `dx` arguments to *`gradient`* and the `bins` argument to *`histogram`*. Some key differences:

| `x` | `isscalar(x)` | `np.ndim(x) == 0` |
|---|---|---|
| PEP 3141 numeric objects (including builtins) | `True` | `True` |
| builtin string and buffer objects | `True` | `True` |
| other builtin objects, like `pathlib.Path`, *Exception*, the result of `re.compile` | `False` | `True` |
| third-party objects like `matplotlib.figure.Figure` | `False` | `True` |
| zero-dimensional numpy arrays | `False` | `True` |
| other numpy arrays | `False` | `False` |
| *list*, *tuple*, and other sequence objects | `False` | `False` |

## Examples

```
>>> np.isscalar(3.1)
True
>>> np.isscalar(np.array(3.1))
False
>>> np.isscalar([3.1])
False
>>> np.isscalar(False)
True
>>> np.isscalar('numpy')
True
```

NumPy supports PEP 3141 numbers:

```
>>> from fractions import Fraction
>>> np.isscalar(Fraction(5, 17))
True
>>> from numbers import Number
>>> np.isscalar(Number())
True
```

## 4.18.4 Logical operations

| | |
|---|---|
| *logical_and*(x1, x2, /[, out, where, …]) | Compute the truth value of x1 AND x2 element-wise. |
| *logical_or*(x1, x2, /[, out, where, casting, …]) | Compute the truth value of x1 OR x2 element-wise. |
| *logical_not*(x, /[, out, where, casting, …]) | Compute the truth value of NOT x element-wise. |
| *logical_xor*(x1, x2, /[, out, where, …]) | Compute the truth value of x1 XOR x2, element-wise. |

`numpy.`**`logical_and`**(*x1*, *x2*, */*, *out=None*, ***, *where=True*, *casting='same_kind'*, *order='K'*, *dtype=None*, *subok=True*[, *signature*, *extobj* ]) **`= <ufunc 'logical_and'>`**

Compute the truth value of x1 AND x2 element-wise.

**Parameters**

**x1, x2**

[array_like] Input arrays. If `x1.shape != x2.shape`, they must be broadcastable to a common shape (which becomes the shape of the output).

**out**

[ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

**where**

[array_like, optional] This condition is broadcast over the input. At locations where the condition is True, the *out* array will be set to the ufunc result. Elsewhere, the *out* array will retain its original value. Note that if an uninitialized *out* array is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.

****kwargs**

For other keyword-only arguments, see the *ufunc docs*.

**Returns**

**y**

[ndarray or bool] Boolean result of the logical AND operation applied to the elements of *x1* and *x2*; the shape is determined by broadcasting. This is a scalar if both *x1* and *x2* are scalars.

**See also:**

*logical_or*, *logical_not*, *logical_xor*, *bitwise_and*

### Examples

```
>>> np.logical_and(True, False)
False
>>> np.logical_and([True, False], [False, False])
array([False, False])
```

```
>>> x = np.arange(5)
>>> np.logical_and(x>1, x<4)
array([False, False,  True,  True, False])
```

`numpy.`**`logical_or`**(*x1*, *x2*, */*, *out=None*, ***, *where=True*, *casting='same_kind'*, *order='K'*, *dtype=None*, *subok=True*[, *signature*, *extobj* ]) **`= <ufunc 'logical_or'>`**

Compute the truth value of x1 OR x2 element-wise.

**Parameters**

**x1, x2**

[array_like] Logical OR is applied to the elements of *x1* and *x2*. If `x1.shape != x2.shape`, they must be broadcastable to a common shape (which becomes the shape of the output).

**out**

[ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

**where**

[array_like, optional] This condition is broadcast over the input. At locations where the condition is True, the *out* array will be set to the ufunc result. Elsewhere, the *out* array will retain its original value. Note that if an uninitialized *out* array is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.

****kwargs**

For other keyword-only arguments, see the *ufunc docs*.

**Returns**

**y**

[ndarray or bool] Boolean result of the logical OR operation applied to the elements of *x1* and *x2*; the shape is determined by broadcasting. This is a scalar if both *x1* and *x2* are scalars.

**See also:**

`logical_and`, `logical_not`, `logical_xor`, `bitwise_or`

## Examples

```
>>> np.logical_or(True, False)
True
>>> np.logical_or([True, False], [False, False])
array([ True, False])
```

```
>>> x = np.arange(5)
>>> np.logical_or(x < 1, x > 3)
array([ True, False, False, False,  True])
```

`numpy.`**`logical_not`**(*x*, */*, *out=None*, ***, *where=True*, *casting='same_kind'*, *order='K'*, *dtype=None*, *subok=True*[, *signature*, *extobj* ]) **`= <ufunc 'logical_not'>`**

Compute the truth value of NOT x element-wise.

**Parameters**

**x**

[array_like] Logical NOT is applied to the elements of *x*.

**out**

[ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

**where**

[array_like, optional] This condition is broadcast over the input. At locations where the condition is True, the *out* array will be set to the ufunc result. Elsewhere, the *out* array will retain its original value. Note that if an uninitialized *out* array is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.

****kwargs**

For other keyword-only arguments, see the *ufunc docs*.

**Returns**

**y**

[bool or ndarray of bool] Boolean result with the same shape as *x* of the NOT operation on elements of *x*. This is a scalar if *x* is a scalar.

**See also:**

`logical_and`, `logical_or`, `logical_xor`

### Examples

```
>>> np.logical_not(3)
False
>>> np.logical_not([True, False, 0, 1])
array([False,  True,  True, False])
```

```
>>> x = np.arange(5)
>>> np.logical_not(x<3)
array([False, False, False,  True,  True])
```

`numpy.`**`logical_xor`**(*x1*, *x2*, */*, *out=None*, ***, *where=True*, *casting='same_kind'*, *order='K'*, *dtype=None*, *subok=True*[, *signature*, *extobj* ]) **`= <ufunc 'logical_xor'>`**

Compute the truth value of x1 XOR x2, element-wise.

**Parameters**

**x1, x2**

[array_like] Logical XOR is applied to the elements of *x1* and *x2*. If `x1.shape != x2.shape`, they must be broadcastable to a common shape (which becomes the shape of the output).

**out**

[ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

**where**

[array_like, optional] This condition is broadcast over the input. At locations where the condition is True, the *out* array will be set to the ufunc result. Elsewhere, the *out* array will retain its original value. Note that if an uninitialized *out* array is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.

****kwargs**

For other keyword-only arguments, see the *ufunc docs*.

**Returns**

**y**

[bool or ndarray of bool] Boolean result of the logical XOR operation applied to the elements of *x1* and *x2*; the shape is determined by broadcasting. This is a scalar if both *x1* and *x2* are scalars.

**See also:**

*`logical_and`*, *`logical_or`*, *`logical_not`*, *`bitwise_xor`*

### Examples

```
>>> np.logical_xor(True, False)
True
>>> np.logical_xor([True, True, False, False], [True, False, True, False])
array([False,  True,  True, False])
```

```
>>> x = np.arange(5)
>>> np.logical_xor(x < 1, x > 3)
array([ True, False, False, False,  True])
```

Simple example showing support of broadcasting

```
>>> np.logical_xor(0, np.eye(2))
array([[ True, False],
       [False,  True]])
```

## 4.18.5 Comparison

| | |
|---|---|
| *`allclose`*(a, b[, rtol, atol, equal_nan]) | Returns True if two arrays are element-wise equal within a tolerance. |
| *`isclose`*(a, b[, rtol, atol, equal_nan]) | Returns a boolean array where two arrays are element-wise equal within a tolerance. |
| *`array_equal`*(a1, a2[, equal_nan]) | True if two arrays have the same shape and elements, False otherwise. |
| *`array_equiv`*(a1, a2) | Returns True if input arrays are shape consistent and all elements equal. |

`numpy.allclose`(*a*, *b*, *rtol=1e-05*, *atol=1e-08*, *equal_nan=False*)

Returns True if two arrays are element-wise equal within a tolerance.

The tolerance values are positive, typically very small numbers. The relative difference (*rtol* * abs(*b*)) and the absolute difference *atol* are added together to compare against the absolute difference between *a* and *b*.

NaNs are treated as equal if they are in the same place and if `equal_nan=True`. Infs are treated as equal if they are in the same place and of the same sign in both arrays.

**Parameters**

**a, b**

[array_like] Input arrays to compare.

**rtol**

[float] The relative tolerance parameter (see Notes).

**atol**

[float] The absolute tolerance parameter (see Notes).

**equal_nan**

[bool] Whether to compare NaN's as equal. If True, NaN's in *a* will be considered equal to NaN's in *b* in the output array.

New in version 1.10.0.

**Returns**

**allclose**

[bool] Returns True if the two arrays are equal within the given tolerance; False otherwise.

**See also:**

*`isclose`*, *`all`*, *`any`*, *`equal`*

## Notes

If the following equation is element-wise True, then allclose returns True.

absolute($a$ - $b$) <= ($atol$ + $rtol$ * absolute($b$))

The above equation is not symmetric in *a* and *b*, so that `allclose(a, b)` might be different from `allclose(b, a)` in some rare cases.

The comparison of *a* and *b* uses standard broadcasting, which means that *a* and *b* need not have the same shape in order for `allclose(a, b)` to evaluate to True. The same is true for *`equal`* but not *`array_equal`*.

### Examples

```
>>> np.allclose([1e10,1e-7], [1.00001e10,1e-8])
False
>>> np.allclose([1e10,1e-8], [1.00001e10,1e-9])
True
>>> np.allclose([1e10,1e-8], [1.0001e10,1e-9])
False
>>> np.allclose([1.0, np.nan], [1.0, np.nan])
False
>>> np.allclose([1.0, np.nan], [1.0, np.nan], equal_nan=True)
True
```

`numpy.`**`isclose`**(*a*, *b*, *rtol=1e-05*, *atol=1e-08*, *equal_nan=False*)

Returns a boolean array where two arrays are element-wise equal within a tolerance.

The tolerance values are positive, typically very small numbers. The relative difference (*rtol* * abs(*b*)) and the absolute difference *atol* are added together to compare against the absolute difference between *a* and *b*.

> **Warning:** The default *atol* is not appropriate for comparing numbers that are much smaller than one (see Notes).

**Parameters**

**a, b**

[array_like] Input arrays to compare.

**rtol**

[float] The relative tolerance parameter (see Notes).

**atol**

[float] The absolute tolerance parameter (see Notes).

**equal_nan**

[bool] Whether to compare NaN's as equal. If True, NaN's in *a* will be considered equal to NaN's in *b* in the output array.

**Returns**

**y**

[array_like] Returns a boolean array of where *a* and *b* are equal within the given tolerance. If both *a* and *b* are scalars, returns a single boolean value.

**See also:**

*`allclose`*

## Notes

New in version 1.7.0.

For finite values, isclose uses the following equation to test whether two floating point values are equivalent.

absolute(*a* - *b*) <= (*atol* + *rtol* * absolute(*b*))

Unlike the built-in `math.isclose`, the above equation is not symmetric in *a* and *b* – it assumes *b* is the reference value – so that *isclose(a, b)* might be different from *isclose(b, a)*. Furthermore, the default value of atol is not zero, and is used to determine what small values should be considered close to zero. The default value is appropriate for expected values of order unity: if the expected values are significantly smaller than one, it can result in false positives. *atol* should be carefully selected for the use case at hand. A zero value for *atol* will result in *False* if either *a* or *b* is zero.

## Examples

```
>>> np.isclose([1e10,1e-7], [1.00001e10,1e-8])
array([ True, False])
>>> np.isclose([1e10,1e-8], [1.00001e10,1e-9])
array([ True, True])
>>> np.isclose([1e10,1e-8], [1.0001e10,1e-9])
array([False,  True])
>>> np.isclose([1.0, np.nan], [1.0, np.nan])
array([ True, False])
>>> np.isclose([1.0, np.nan], [1.0, np.nan], equal_nan=True)
array([ True, True])
>>> np.isclose([1e-8, 1e-7], [0.0, 0.0])
array([ True, False])
>>> np.isclose([1e-100, 1e-7], [0.0, 0.0], atol=0.0)
array([False, False])
>>> np.isclose([1e-10, 1e-10], [1e-20, 0.0])
array([ True,  True])
>>> np.isclose([1e-10, 1e-10], [1e-20, 0.999999e-10], atol=0.0)
array([False,  True])
```

`numpy.`**`array_equal`**(*a1*, *a2*, *equal_nan=False*)

True if two arrays have the same shape and elements, False otherwise.

**Parameters**

**a1, a2**

[array_like] Input arrays.

**equal_nan**

[bool] Whether to compare NaN's as equal. If the dtype of a1 and a2 is complex, values will be considered equal if either the real or the imaginary component of a given value is `nan`.

New in version 1.19.0.

**Returns**

**b**

[bool] Returns True if the arrays are equal.

**See also:**

*`allclose`*

Returns True if two arrays are element-wise equal within a tolerance.

*`array_equiv`*

Returns True if input arrays are shape consistent and all elements equal.

### Examples

```
>>> np.array_equal([1, 2], [1, 2])
True
>>> np.array_equal(np.array([1, 2]), np.array([1, 2]))
True
>>> np.array_equal([1, 2], [1, 2, 3])
False
>>> np.array_equal([1, 2], [1, 4])
False
>>> a = np.array([1, np.nan])
>>> np.array_equal(a, a)
False
>>> np.array_equal(a, a, equal_nan=True)
True
```

When `equal_nan` is True, complex values with nan components are considered equal if either the real *or* the imaginary components are nan.

```
>>> a = np.array([1 + 1j])
>>> b = a.copy()
>>> a.real = np.nan
>>> b.imag = np.nan
>>> np.array_equal(a, b, equal_nan=True)
True
```

`numpy.`**`array_equiv`**(*a1*, *a2*)

Returns True if input arrays are shape consistent and all elements equal.

Shape consistent means they are either the same shape, or one input array can be broadcasted to create the same shape as the other one.

**Parameters**

**a1, a2**

[array_like] Input arrays.

**Returns**

**out**

[bool] True if equivalent, False otherwise.

### Examples

```
>>> np.array_equiv([1, 2], [1, 2])
True
>>> np.array_equiv([1, 2], [1, 3])
False
```

Showing the shape equivalence:

```
>>> np.array_equiv([1, 2], [[1, 2], [1, 2]])
True
>>> np.array_equiv([1, 2], [[1, 2, 1, 2], [1, 2, 1, 2]])
False
```

```
>>> np.array_equiv([1, 2], [[1, 2], [1, 3]])
False
```

| | |
|---|---|
| `greater`(x1, x2, /[, out, where, casting, …]) | Return the truth value of (x1 > x2) element-wise. |
| `greater_equal`(x1, x2, /[, out, where, …]) | Return the truth value of (x1 >= x2) element-wise. |
| `less`(x1, x2, /[, out, where, casting, …]) | Return the truth value of (x1 < x2) element-wise. |
| `less_equal`(x1, x2, /[, out, where, casting, …]) | Return the truth value of (x1 =< x2) element-wise. |
| `equal`(x1, x2, /[, out, where, casting, …]) | Return (x1 == x2) element-wise. |
| `not_equal`(x1, x2, /[, out, where, casting, …]) | Return (x1 != x2) element-wise. |

`numpy.`**`greater`**(*x1*, *x2*, */*, *out=None*, ***, *where=True*, *casting='same_kind'*, *order='K'*, *dtype=None*, *subok=True*[, *signature*, *extobj*]) **`= <ufunc 'greater'>`**

Return the truth value of (x1 > x2) element-wise.

**Parameters**

**x1, x2**

[array_like] Input arrays. If `x1.shape != x2.shape`, they must be broadcastable to a common shape (which becomes the shape of the output).

**out**

[ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

**where**

[array_like, optional] This condition is broadcast over the input. At locations where the condition is True, the *out* array will be set to the ufunc result. Elsewhere, the *out* array will retain its original value. Note that if an uninitialized *out* array is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.

****kwargs**

For other keyword-only arguments, see the *ufunc docs*.

**Returns**

**out**

[ndarray or scalar] Output array, element-wise comparison of *x1* and *x2*. Typically of type bool, unless `dtype=object` is passed. This is a scalar if both *x1* and *x2* are scalars.

**See also:**

`greater_equal`, `less`, `less_equal`, `equal`, `not_equal`

## Examples

```
>>> np.greater([4,2],[2,2])
array([ True, False])
```

If the inputs are ndarrays, then np.greater is equivalent to '>'.

```
>>> a = np.array([4,2])
>>> b = np.array([2,2])
>>> a > b
array([ True, False])
```

`numpy.`**`greater_equal`**(*x1*, *x2*, */*, *out=None*, ***, *where=True*, *casting='same_kind'*, *order='K'*, *dtype=None*, *subok=True*[, *signature*, *extobj* ]) **`= <ufunc 'greater_equal'>`**

Return the truth value of (x1 >= x2) element-wise.

**Parameters**

**x1, x2**

[array_like] Input arrays. If `x1.shape != x2.shape`, they must be broadcastable to a common shape (which becomes the shape of the output).

**out**

[ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

**where**

[array_like, optional] This condition is broadcast over the input. At locations where the condition is True, the *out* array will be set to the ufunc result. Elsewhere, the *out* array will retain its original value. Note that if an uninitialized *out* array is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.

****kwargs**

For other keyword-only arguments, see the *ufunc docs*.

**Returns**

**out**

[bool or ndarray of bool] Output array, element-wise comparison of *x1* and *x2*. Typically of type bool, unless `dtype=object` is passed. This is a scalar if both *x1* and *x2* are scalars.

**See also:**

`greater`, `less`, `less_equal`, `equal`, `not_equal`

### Examples

```
>>> np.greater_equal([4, 2, 1], [2, 2, 2])
array([ True, True, False])
```

`numpy.`**`less`**(*x1*, *x2*, */*, *out=None*, ***, *where=True*, *casting='same_kind'*, *order='K'*, *dtype=None*, *subok=True*[, *signature*, *extobj*]) = **`<ufunc 'less'>`**

Return the truth value of (x1 < x2) element-wise.

**Parameters**

**x1, x2**

[array_like] Input arrays. If `x1.shape != x2.shape`, they must be broadcastable to a common shape (which becomes the shape of the output).

**out**

[ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

**where**

[array_like, optional] This condition is broadcast over the input. At locations where the condition is True, the *out* array will be set to the ufunc result. Elsewhere, the *out* array will retain its original value. Note that if an uninitialized *out* array is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.

****kwargs**

For other keyword-only arguments, see the *ufunc docs*.

**Returns**

**out**

[ndarray or scalar] Output array, element-wise comparison of *x1* and *x2*. Typically of type bool, unless `dtype=object` is passed. This is a scalar if both *x1* and *x2* are scalars.

**See also:**

*`greater`*, *`less_equal`*, *`greater_equal`*, *`equal`*, *`not_equal`*

### Examples

```
>>> np.less([1, 2], [2, 2])
array([ True, False])
```

`numpy.`**`less_equal`**(*x1*, *x2*, */*, *out=None*, ***, *where=True*, *casting='same_kind'*, *order='K'*, *dtype=None*, *subok=True*[, *signature*, *extobj*]) = **`<ufunc 'less_equal'>`**

Return the truth value of (x1 =< x2) element-wise.

**Parameters**

**x1, x2**

[array_like] Input arrays. If `x1.shape != x2.shape`, they must be broadcastable to a common shape (which becomes the shape of the output).

**out**

[ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

**where**

[array_like, optional] This condition is broadcast over the input. At locations where the condition is True, the *out* array will be set to the ufunc result. Elsewhere, the *out* array will retain its original value. Note that if an uninitialized *out* array is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.

****kwargs**

For other keyword-only arguments, see the *ufunc docs*.

**Returns**

**out**

[ndarray or scalar] Output array, element-wise comparison of *x1* and *x2*. Typically of type bool, unless `dtype=object` is passed. This is a scalar if both *x1* and *x2* are scalars.

**See also:**

`greater`, `less`, `greater_equal`, `equal`, `not_equal`

### Examples

```
>>> np.less_equal([4, 2, 1], [2, 2, 2])
array([False,  True,  True])
```

`numpy.`**`equal`** (*x1, x2, /, out=None, *, where=True, casting='same_kind', order='K', dtype=None, subok=True*[, *signature, extobj*]) **= <ufunc 'equal'>**

Return (x1 == x2) element-wise.

**Parameters**

**x1, x2**

[array_like] Input arrays. If `x1.shape != x2.shape`, they must be broadcastable to a common shape (which becomes the shape of the output).

**out**

[ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

**where**

[array_like, optional] This condition is broadcast over the input. At locations where the condition is True, the *out* array will be set to the ufunc result. Elsewhere, the *out* array will retain

its original value. Note that if an uninitialized *out* array is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.

****kwargs**

For other keyword-only arguments, see the *ufunc docs*.

**Returns**

**out**

[ndarray or scalar] Output array, element-wise comparison of *x1* and *x2*. Typically of type bool, unless `dtype=object` is passed. This is a scalar if both *x1* and *x2* are scalars.

**See also:**

`not_equal`, `greater_equal`, `less_equal`, `greater`, `less`

### Examples

```
>>> np.equal([0, 1, 3], np.arange(3))
array([ True,  True, False])
```

What is compared are values, not types. So an int (1) and an array of length one can evaluate as True:

```
>>> np.equal(1, np.ones(1))
array([ True])
```

`numpy.`**`not_equal`**(*x1*, *x2*, */*, *out=None*, ***, *where=True*, *casting='same_kind'*, *order='K'*, *dtype=None*, *subok=True*[, *signature*, *extobj* ]) **`= <ufunc 'not_equal'>`**

Return (x1 != x2) element-wise.

**Parameters**

**x1, x2**

[array_like] Input arrays. If `x1.shape != x2.shape`, they must be broadcastable to a common shape (which becomes the shape of the output).

**out**

[ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

**where**

[array_like, optional] This condition is broadcast over the input. At locations where the condition is True, the *out* array will be set to the ufunc result. Elsewhere, the *out* array will retain its original value. Note that if an uninitialized *out* array is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.

****kwargs**

For other keyword-only arguments, see the *ufunc docs*.

**Returns**

**out**

[ndarray or scalar] Output array, element-wise comparison of *x1* and *x2*. Typically of type bool, unless `dtype=object` is passed. This is a scalar if both *x1* and *x2* are scalars.

**See also:**

*equal*, *greater*, *greater_equal*, *less*, *less_equal*

### Examples

```
>>> np.not_equal([1.,2.], [1., 3.])
array([False,  True])
>>> np.not_equal([1, 2], [[1, 3],[1, 4]])
array([[False,  True],
       [False,  True]])
```

## 4.19 Mathematical functions

### 4.19.1 Trigonometric functions

| | |
|---|---|
| *sin*(x, /[, out, where, casting, order, …]) | Trigonometric sine, element-wise. |
| *cos*(x, /[, out, where, casting, order, …]) | Cosine element-wise. |
| *tan*(x, /[, out, where, casting, order, …]) | Compute tangent element-wise. |
| *arcsin*(x, /[, out, where, casting, order, …]) | Inverse sine, element-wise. |
| *arccos*(x, /[, out, where, casting, order, …]) | Trigonometric inverse cosine, element-wise. |
| *arctan*(x, /[, out, where, casting, order, …]) | Trigonometric inverse tangent, element-wise. |
| *hypot*(x1, x2, /[, out, where, casting, …]) | Given the “legs” of a right triangle, return its hypotenuse. |
| *arctan2*(x1, x2, /[, out, where, casting, …]) | Element-wise arc tangent of `x1/x2` choosing the quadrant correctly. |
| *degrees*(x, /[, out, where, casting, order, …]) | Convert angles from radians to degrees. |
| *radians*(x, /[, out, where, casting, order, …]) | Convert angles from degrees to radians. |
| *unwrap*(p[, discont, axis]) | Unwrap by changing deltas between values to 2*pi complement. |
| *deg2rad*(x, /[, out, where, casting, order, …]) | Convert angles from degrees to radians. |
| *rad2deg*(x, /[, out, where, casting, order, …]) | Convert angles from radians to degrees. |

`numpy.`**`sin`**(*x*, */*, *out=None*, ***, *where=True*, *casting='same_kind'*, *order='K'*, *dtype=None*, *subok=True*[, *signature*, *extobj*]) **`= <ufunc 'sin'>`**

Trigonometric sine, element-wise.

**Parameters**

**x**

[array_like] Angle, in radians ($2\pi$ rad equals 360 degrees).

**out**

[ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

**where**

[array_like, optional] This condition is broadcast over the input. At locations where the condition is True, the *out* array will be set to the ufunc result. Elsewhere, the *out* array will retain its original value. Note that if an uninitialized *out* array is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.

****kwargs**

For other keyword-only arguments, see the *ufunc docs*.

**Returns**

**y**

[array_like] The sine of each element of x. This is a scalar if *x* is a scalar.

**See also:**

`arcsin`, `sinh`, `cos`

## Notes

The sine is one of the fundamental functions of trigonometry (the mathematical study of triangles). Consider a circle of radius 1 centered on the origin. A ray comes in from the $+x$ axis, makes an angle at the origin (measured counter-clockwise from that axis), and departs from the origin. The $y$ coordinate of the outgoing ray's intersection with the unit circle is the sine of that angle. It ranges from -1 for $x = 3\pi/2$ to +1 for $\pi/2$. The function has zeroes where the angle is a multiple of $\pi$. Sines of angles between $\pi$ and $2\pi$ are negative. The numerous properties of the sine and related functions are included in any standard trigonometry text.

## Examples

Print sine of one angle:

```
>>> np.sin(np.pi/2.)
1.0
```

Print sines of an array of angles given in degrees:

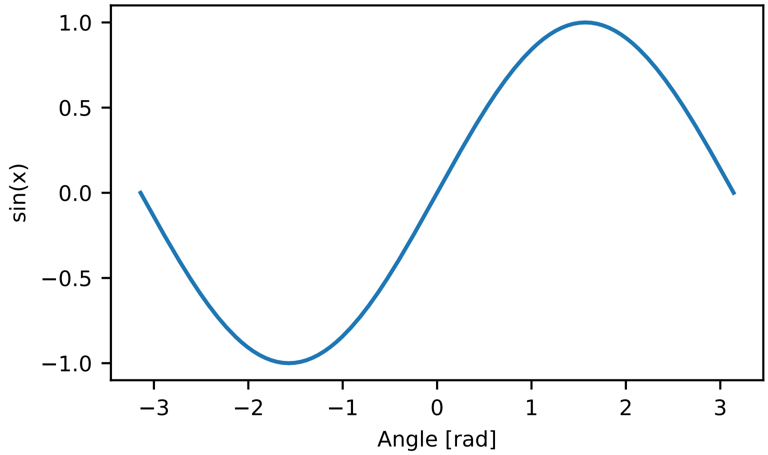
```
>>> np.sin(np.array((0., 30., 45., 60., 90.)) * np.pi / 180. )
array([ 0.        ,  0.5       ,  0.70710678,  0.8660254 ,  1.        ])
```

Plot the sine function:

```
>>> import matplotlib.pylab as plt
>>> x = np.linspace(-np.pi, np.pi, 201)
>>> plt.plot(x, np.sin(x))
>>> plt.xlabel('Angle [rad]')
>>> plt.ylabel('sin(x)')
>>> plt.axis('tight')
>>> plt.show()
```

`numpy.`**`cos`**(*x*, */*, *out=None*, ***, *where=True*, *casting='same_kind'*, *order='K'*, *dtype=None*, *subok=True*[, *signature*, *extobj*]) **= <ufunc 'cos'>**

Cosine element-wise.



**Parameters**

**x**

[array_like] Input array in radians.

**out**

[ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

**where**

[array_like, optional] This condition is broadcast over the input. At locations where the condition is True, the *out* array will be set to the ufunc result. Elsewhere, the *out* array will retain its original value. Note that if an uninitialized *out* array is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.

****kwargs**

For other keyword-only arguments, see the *ufunc docs*.

**Returns**

**y**

[ndarray] The corresponding cosine values. This is a scalar if *x* is a scalar.

### Notes

If *out* is provided, the function writes the result into it, and returns a reference to *out*. (See Examples)

### References

M. Abramowitz and I. A. Stegun, Handbook of Mathematical Functions. New York, NY: Dover, 1972.

### Examples

```
>>> np.cos(np.array([0, np.pi/2, np.pi]))
array([  1.00000000e+00,   6.12303177e-17,  -1.00000000e+00])
>>>
>>> # Example of providing the optional output parameter
>>> out1 = np.array([0], dtype='d')
>>> out2 = np.cos([0.1], out1)
>>> out2 is out1
True
>>>
>>> # Example of ValueError due to provision of shape mis-matched `out`
>>> np.cos(np.zeros((3,3)),np.zeros((2,2)))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: operands could not be broadcast together with shapes (3,3) (2,2)
```

`numpy.`**`tan`**(*x*, */*, *out=None*, ***, *where=True*, *casting='same_kind'*, *order='K'*, *dtype=None*, *subok=True*[, *signature*, *extobj* ]) **`= <ufunc 'tan'>`**

Compute tangent element-wise.

Equivalent to `np.sin(x)/np.cos(x)` element-wise.

**Parameters**

**x**

[array_like] Input array.

**out**

[ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

**where**

[array_like, optional] This condition is broadcast over the input. At locations where the condition is True, the *out* array will be set to the ufunc result. Elsewhere, the *out* array will retain its original value. Note that if an uninitialized *out* array is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.

****kwargs**

For other keyword-only arguments, see the *ufunc docs*.

**Returns**

**y**

[ndarray] The corresponding tangent values. This is a scalar if *x* is a scalar.

## Notes

If *out* is provided, the function writes the result into it, and returns a reference to *out*. (See Examples)

## References

M. Abramowitz and I. A. Stegun, Handbook of Mathematical Functions. New York, NY: Dover, 1972.

## Examples

```
>>> from math import pi
>>> np.tan(np.array([-pi,pi/2,pi]))
array([  1.22460635e-16,   1.63317787e+16,  -1.22460635e-16])
>>>
>>> # Example of providing the optional output parameter illustrating
>>> # that what is returned is a reference to said parameter
>>> out1 = np.array([0], dtype='d')
>>> out2 = np.cos([0.1], out1)
>>> out2 is out1
True
>>>
>>> # Example of ValueError due to provision of shape mis-matched `out`
>>> np.cos(np.zeros((3,3)),np.zeros((2,2)))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: operands could not be broadcast together with shapes (3,3) (2,2)
```

`numpy.`**`arcsin`**(*x*, */*, *out=None*, ***, *where=True*, *casting='same_kind'*, *order='K'*, *dtype=None*, *subok=True*[, *signature*, *extobj* ]) **`= <ufunc 'arcsin'>`**

Inverse sine, element-wise.

**Parameters**

**x**

[array_like] *y*-coordinate on the unit circle.

**out**

[ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

**where**

[array_like, optional] This condition is broadcast over the input. At locations where the condition is True, the *out* array will be set to the ufunc result. Elsewhere, the *out* array will retain its original value. Note that if an uninitialized *out* array is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.

****kwargs**

For other keyword-only arguments, see the *ufunc docs*.

**Returns**

**angle**

[ndarray] The inverse sine of each element in *x*, in radians and in the closed interval `[-pi/2, pi/2]`. This is a scalar if *x* is a scalar.

**See also:**

`sin`, `cos`, `arccos`, `tan`, `arctan`, `arctan2`, `emath.arcsin`

### Notes

`arcsin` is a multivalued function: for each *x* there are infinitely many numbers *z* such that $sin(z) = x$. The convention is to return the angle *z* whose real part lies in [-pi/2, pi/2].

For real-valued input data types, *arcsin* always returns real output. For each value that cannot be expressed as a real number or infinity, it yields `nan` and sets the *invalid* floating point error flag.

For complex-valued input, `arcsin` is a complex analytic function that has, by convention, the branch cuts [-inf, -1] and [1, inf] and is continuous from above on the former and from below on the latter.

The inverse sine is also known as *asin* or sin^{-1}.

### References

Abramowitz, M. and Stegun, I. A., *Handbook of Mathematical Functions*, 10th printing, New York: Dover, 1964, pp. 79ff. http://www.math.sfu.ca/~cbm/aands/

### Examples

```
>>> np.arcsin(1)     # pi/2
1.5707963267948966
>>> np.arcsin(-1)    # -pi/2
-1.5707963267948966
>>> np.arcsin(0)
0.0
```

`numpy.`**`arccos`**(*x*, */*, *out=None*, ***, *where=True*, *casting='same_kind'*, *order='K'*, *dtype=None*, *subok=True*[, *signature*, *extobj* ]) **`= <ufunc 'arccos'>`**

Trigonometric inverse cosine, element-wise.

The inverse of `cos` so that, if `y = cos(x)`, then `x = arccos(y)`.

**Parameters**

**x**

[array_like] *x*-coordinate on the unit circle. For real arguments, the domain is [-1, 1].

**out**

[ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

**where**

[array_like, optional] This condition is broadcast over the input. At locations where the condition is True, the *out* array will be set to the ufunc result. Elsewhere, the *out* array will retain its original value. Note that if an uninitialized *out* array is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.

****kwargs**

For other keyword-only arguments, see the *ufunc docs*.

**Returns**

**angle**

[ndarray] The angle of the ray intersecting the unit circle at the given *x*-coordinate in radians [0, pi]. This is a scalar if *x* is a scalar.

**See also:**

`cos`, `arctan`, `arcsin`, `emath.arccos`

## Notes

`arccos` is a multivalued function: for each *x* there are infinitely many numbers *z* such that *cos(z) = x*. The convention is to return the angle *z* whose real part lies in *[0, pi]*.

For real-valued input data types, `arccos` always returns real output. For each value that cannot be expressed as a real number or infinity, it yields `nan` and sets the *invalid* floating point error flag.

For complex-valued input, `arccos` is a complex analytic function that has branch cuts *[-inf, -1]* and *[1, inf]* and is continuous from above on the former and from below on the latter.

The inverse `cos` is also known as *acos* or cos^-1.

## References

M. Abramowitz and I.A. Stegun, "Handbook of Mathematical Functions", 10th printing, 1964, pp. 79. http://www.math.sfu.ca/~cbm/aands/

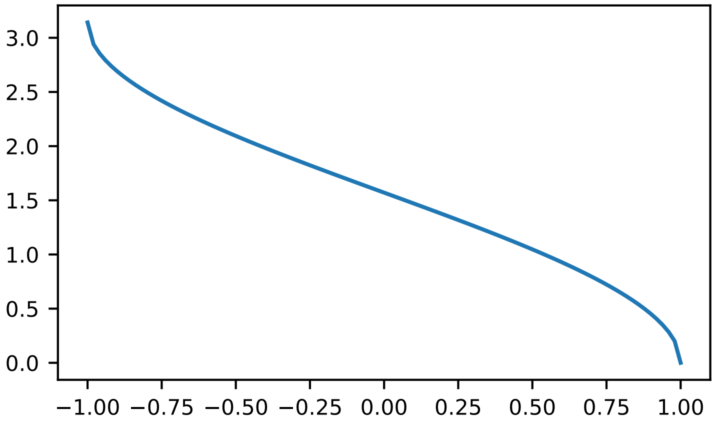
### Examples

We expect the arccos of 1 to be 0, and of -1 to be pi:

```
>>> np.arccos([1, -1])
array([ 0.        ,  3.14159265])
```

Plot arccos:

```
>>> import matplotlib.pyplot as plt
>>> x = np.linspace(-1, 1, num=100)
>>> plt.plot(x, np.arccos(x))
>>> plt.axis('tight')
>>> plt.show()
```



`numpy.`**`arctan`**(*x*, */*, *out=None*, ***, *where=True*, *casting='same_kind'*, *order='K'*, *dtype=None*, *subok=True*[, *signature*, *extobj*]) **`= <ufunc 'arctan'>`**

Trigonometric inverse tangent, element-wise.

The inverse of tan, so that if `y = tan(x)` then `x = arctan(y)`.

**Parameters**

**x**

[array_like]

**out**

[ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

**where**

[array_like, optional] This condition is broadcast over the input. At locations where the condition is True, the *out* array will be set to the ufunc result. Elsewhere, the *out* array will retain

its original value. Note that if an uninitialized *out* array is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.

****kwargs**

For other keyword-only arguments, see the *ufunc docs*.

**Returns**

**out**

[ndarray or scalar] Out has the same shape as *x*. Its real part is in `[-pi/2, pi/2]` (`arctan(+/-inf)` returns `+/-pi/2`). This is a scalar if *x* is a scalar.

**See also:**

***`arctan2`***

The "four quadrant" arctan of the angle formed by (*x*, *y*) and the positive *x*-axis.

***`angle`***

Argument of complex values.

## Notes

`arctan` is a multi-valued function: for each *x* there are infinitely many numbers *z* such that tan(*z*) = *x*. The convention is to return the angle *z* whose real part lies in [-pi/2, pi/2].

For real-valued input data types, `arctan` always returns real output. For each value that cannot be expressed as a real number or infinity, it yields `nan` and sets the *invalid* floating point error flag.

For complex-valued input, `arctan` is a complex analytic function that has [*1j, infj*] and [*-1j, -infj*] as branch cuts, and is continuous from the left on the former and from the right on the latter.

The inverse tangent is also known as *atan* or tan^{-1}.

## References

Abramowitz, M. and Stegun, I. A., *Handbook of Mathematical Functions*, 10th printing, New York: Dover, 1964, pp. 79. http://www.math.sfu.ca/~cbm/aands/

## Examples

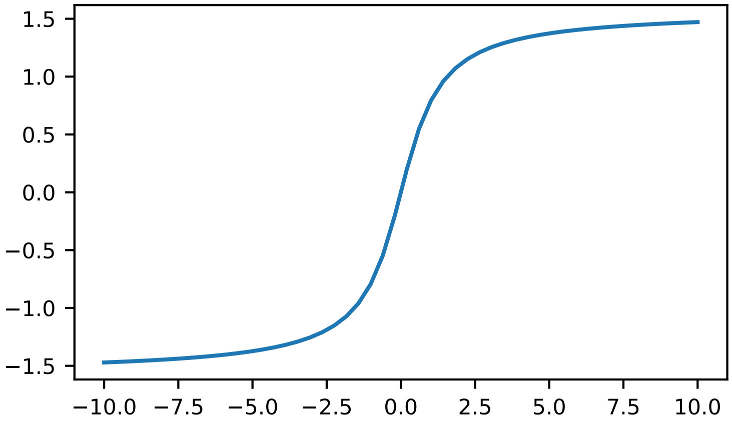
We expect the arctan of 0 to be 0, and of 1 to be pi/4:

```
>>> np.arctan([0, 1])
array([ 0.        ,  0.78539816])
```

```
>>> np.pi/4
0.78539816339744828
```

Plot arctan:

```
>>> import matplotlib.pyplot as plt
>>> x = np.linspace(-10, 10)
>>> plt.plot(x, np.arctan(x))
>>> plt.axis('tight')
>>> plt.show()
```



`numpy.`**`hypot`**(*x1*, *x2*, /, *out=None*, *, *where=True*, *casting='same_kind'*, *order='K'*, *dtype=None*, *subok=True*[, *signature*, *extobj* ]) **`= <ufunc 'hypot'>`**

Given the "legs" of a right triangle, return its hypotenuse.

Equivalent to `sqrt(x1**2 + x2**2)`, element-wise. If *x1* or *x2* is scalar_like (i.e., unambiguously cast-able to a scalar type), it is broadcast for use with each element of the other argument. (See Examples)

**Parameters**

**x1, x2**

[array_like] Leg of the triangle(s). If `x1.shape != x2.shape`, they must be broadcastable to a common shape (which becomes the shape of the output).

**out**

[ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

**where**

[array_like, optional] This condition is broadcast over the input. At locations where the condition is True, the *out* array will be set to the ufunc result. Elsewhere, the *out* array will retain its original value. Note that if an uninitialized *out* array is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.

****kwargs**

For other keyword-only arguments, see the *ufunc docs*.

**Returns**

**z**

[ndarray] The hypotenuse of the triangle(s). This is a scalar if both *x1* and *x2* are scalars.

## Examples

```
>>> np.hypot(3*np.ones((3, 3)), 4*np.ones((3, 3)))
array([[ 5.,  5.,  5.],
       [ 5.,  5.,  5.],
       [ 5.,  5.,  5.]])
```

Example showing broadcast of scalar_like argument:

```
>>> np.hypot(3*np.ones((3, 3)), [4])
array([[ 5.,  5.,  5.],
       [ 5.,  5.,  5.],
       [ 5.,  5.,  5.]])
```

`numpy.`**`arctan2`**(*x1*, *x2*, */*, *out=None*, ***, *where=True*, *casting='same_kind'*, *order='K'*, *dtype=None*, *subok=True*[, *signature*, *extobj* ]) **`= <ufunc 'arctan2'>`**

Element-wise arc tangent of `x1/x2` choosing the quadrant correctly.

The quadrant (i.e., branch) is chosen so that `arctan2(x1, x2)` is the signed angle in radians between the ray ending at the origin and passing through the point (1,0), and the ray ending at the origin and passing through the point (*x2*, *x1*). (Note the role reversal: the "*y*-coordinate" is the first function parameter, the "*x*-coordinate" is the second.) By IEEE convention, this function is defined for *x2* = +/-0 and for either or both of *x1* and *x2* = +/-inf (see Notes for specific values).

This function is not defined for complex-valued arguments; for the so-called argument of complex values, use *`angle`*.

**Parameters**

**x1**

[array_like, real-valued] *y*-coordinates.

**x2**

[array_like, real-valued] *x*-coordinates. If `x1.shape != x2.shape`, they must be broadcastable to a common shape (which becomes the shape of the output).

**out**

[ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

**where**

[array_like, optional] This condition is broadcast over the input. At locations where the condition is True, the *out* array will be set to the ufunc result. Elsewhere, the *out* array will retain its original value. Note that if an uninitialized *out* array is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.

****kwargs**

For other keyword-only arguments, see the *ufunc docs*.

**Returns**

**angle**

[ndarray] Array of angles in radians, in the range `[-pi, pi]`. This is a scalar if both *x1* and *x2* are scalars.

**See also:**

`arctan`, `tan`, `angle`

## Notes

*arctan2* is identical to the *atan2* function of the underlying C library. The following special values are defined in the C standard: [1]

| *x1* | *x2* | *arctan2(x1,x2)* |
|---|---|---|
| +/- 0 | +0 | +/- 0 |
| +/- 0 | -0 | +/- pi |
| > 0 | +/-inf | +0 / +pi |
| < 0 | +/-inf | -0 / -pi |
| +/-inf | +inf | +/- (pi/4) |
| +/-inf | -inf | +/- (3*pi/4) |

Note that +0 and -0 are distinct floating point numbers, as are +inf and -inf.

## References

[1]

## Examples

Consider four points in different quadrants:

```
>>> x = np.array([-1, +1, +1, -1])
>>> y = np.array([-1, -1, +1, +1])
>>> np.arctan2(y, x) * 180 / np.pi
array([-135.,  -45.,   45.,  135.])
```

Note the order of the parameters. `arctan2` is defined also when *x2* = 0 and at several other special points, obtaining values in the range `[-pi, pi]`:

```
>>> np.arctan2([1., -1.], [0., 0.])
array([ 1.57079633, -1.57079633])
>>> np.arctan2([0., 0., np.inf], [+0., -0., np.inf])
array([ 0.        ,  3.14159265,  0.78539816])
```

`numpy.`**`degrees`**(*x*, */*, *out=None*, ***, *where=True*, *casting='same_kind'*, *order='K'*, *dtype=None*, *subok=True*[, *signature*, *extobj*]) **`= <ufunc 'degrees'>`**

Convert angles from radians to degrees.

**Parameters**

**x**

[array_like] Input array in radians.

**out**

[ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

**where**

[array_like, optional] This condition is broadcast over the input. At locations where the condition is True, the *out* array will be set to the ufunc result. Elsewhere, the *out* array will retain its original value. Note that if an uninitialized *out* array is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.

****kwargs**

For other keyword-only arguments, see the *ufunc docs*.

**Returns**

**y**

[ndarray of floats] The corresponding degree values; if *out* was supplied this is a reference to it. This is a scalar if *x* is a scalar.

**See also:**

***`rad2deg`***

equivalent function

### Examples

Convert a radian array to degrees

```
>>> rad = np.arange(12.)*np.pi/6
>>> np.degrees(rad)
array([   0.,   30.,   60.,   90.,  120.,  150.,  180.,  210.,  240.,
        270.,  300.,  330.])
```

```
>>> out = np.zeros((rad.shape))
>>> r = np.degrees(rad, out)
>>> np.all(r == out)
True
```

`numpy.`**`radians`**(*x*, */*, *out=None*, *, *where=True*, *casting='same_kind'*, *order='K'*, *dtype=None*, *subok=True*[, *signature*, *extobj*]) **`= <ufunc 'radians'>`**

Convert angles from degrees to radians.

**Parameters**

**x**

[array_like] Input array in degrees.

**out**

[ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

**where**

[array_like, optional] This condition is broadcast over the input. At locations where the condition is True, the *out* array will be set to the ufunc result. Elsewhere, the *out* array will retain its original value. Note that if an uninitialized *out* array is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.

****kwargs**

For other keyword-only arguments, see the *ufunc docs*.

**Returns**

**y**

[ndarray] The corresponding radian values. This is a scalar if *x* is a scalar.

**See also:**

**`deg2rad`**

equivalent function

## Examples

Convert a degree array to radians

```
>>> deg = np.arange(12.) * 30.
>>> np.radians(deg)
array([ 0.        ,  0.52359878,  1.04719755,  1.57079633,  2.0943951 ,
        2.61799388,  3.14159265,  3.66519143,  4.1887902 ,  4.71238898,
        5.23598776,  5.75958653])
```

```
>>> out = np.zeros((deg.shape))
>>> ret = np.radians(deg, out)
>>> ret is out
True
```

`numpy.`**`unwrap`**(*p*, *discont=3.141592653589793*, *axis=-1*)

Unwrap by changing deltas between values to 2*pi complement.

Unwrap radian phase *p* by changing absolute jumps greater than *discont* to their 2*pi complement along the given axis.

**Parameters**

**p**

[array_like] Input array.

**discont**

[float, optional] Maximum discontinuity between values, default is `pi`.

**axis**

[int, optional] Axis along which unwrap will operate, default is the last axis.

**Returns**

**out**

[ndarray] Output array.

**See also:**

*rad2deg*, *deg2rad*

## Notes

If the discontinuity in *p* is smaller than `pi`, but larger than *discont*, no unwrapping is done because taking the 2*pi complement would only make the discontinuity larger.

## Examples

```
>>> phase = np.linspace(0, np.pi, num=5)
>>> phase[3:] += np.pi
>>> phase
array([ 0.        ,  0.78539816,  1.57079633,  5.49778714,  6.28318531]) # may␣
↪vary
>>> np.unwrap(phase)
array([ 0.        ,  0.78539816,  1.57079633, -0.78539816,  0.        ]) # may␣
↪vary
```

`numpy.`**`deg2rad`**(*x*, */*, *out=None*, ***, *where=True*, *casting='same_kind'*, *order='K'*, *dtype=None*, *subok=True*[, *signature*, *extobj* ]) **`= <ufunc 'deg2rad'>`**

Convert angles from degrees to radians.

**Parameters**

**x**

[array_like] Angles in degrees.

**out**

[ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

**where**

[array_like, optional] This condition is broadcast over the input. At locations where the condition is True, the *out* array will be set to the ufunc result. Elsewhere, the *out* array will retain its original value. Note that if an uninitialized *out* array is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.

****kwargs**

For other keyword-only arguments, see the *ufunc docs*.

**Returns**

**y**

[ndarray] The corresponding angle in radians. This is a scalar if *x* is a scalar.

**See also:**

**`rad2deg`**

Convert angles from radians to degrees.

**`unwrap`**

Remove large jumps in angle by wrapping.

## Notes

New in version 1.3.0.

`deg2rad(x)` is `x * pi / 180`.

## Examples

```
>>> np.deg2rad(180)
3.1415926535897931
```

`numpy.`**`rad2deg`**`(`*x*, */*, *out=None*, ***, *where=True*, *casting='same_kind'*, *order='K'*, *dtype=None*, *subok=True*[, *signature*, *extobj*]`)` **`= <ufunc 'rad2deg'>`**

Convert angles from radians to degrees.

**Parameters**

**x**

[array_like] Angle in radians.

**out**

[ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

**where**

[array_like, optional] This condition is broadcast over the input. At locations where the condition is True, the *out* array will be set to the ufunc result. Elsewhere, the *out* array will retain its original value. Note that if an uninitialized *out* array is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.

****kwargs**

For other keyword-only arguments, see the *ufunc docs*.

**Returns**

**y**

[ndarray] The corresponding angle in degrees. This is a scalar if *x* is a scalar.

**See also:**

**`deg2rad`**

Convert angles from degrees to radians.

**`unwrap`**

Remove large jumps in angle by wrapping.

### Notes

New in version 1.3.0.

rad2deg(x) is `180 * x / pi`.

### Examples

```
>>> np.rad2deg(np.pi/2)
90.0
```

## 4.19.2 Hyperbolic functions

| | |
|---|---|
| `sinh`(x, /[, out, where, casting, order, …]) | Hyperbolic sine, element-wise. |
| `cosh`(x, /[, out, where, casting, order, …]) | Hyperbolic cosine, element-wise. |
| `tanh`(x, /[, out, where, casting, order, …]) | Compute hyperbolic tangent element-wise. |
| `arcsinh`(x, /[, out, where, casting, order, …]) | Inverse hyperbolic sine element-wise. |
| `arccosh`(x, /[, out, where, casting, order, …]) | Inverse hyperbolic cosine, element-wise. |
| `arctanh`(x, /[, out, where, casting, order, …]) | Inverse hyperbolic tangent element-wise. |

`numpy.`**`sinh`**(*x*, */*, *out=None*, ***, *where=True*, *casting='same_kind'*, *order='K'*, *dtype=None*, *subok=True*[, *signature*, *extobj*]) **`= <ufunc 'sinh'>`**

Hyperbolic sine, element-wise.

Equivalent to `1/2 * (np.exp(x) - np.exp(-x))` or `-1j * np.sin(1j*x)`.

**Parameters**

**x**

[array_like] Input array.

**out**

[ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

**where**

[array_like, optional] This condition is broadcast over the input. At locations where the condition is True, the *out* array will be set to the ufunc result. Elsewhere, the *out* array will retain its original value. Note that if an uninitialized *out* array is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.

****kwargs**

For other keyword-only arguments, see the *ufunc docs*.

**Returns**

**y**

[ndarray] The corresponding hyperbolic sine values. This is a scalar if *x* is a scalar.

### Notes

If *out* is provided, the function writes the result into it, and returns a reference to *out*. (See Examples)

### References

M. Abramowitz and I. A. Stegun, Handbook of Mathematical Functions. New York, NY: Dover, 1972, pg. 83.

### Examples

```
>>> np.sinh(0)
0.0
>>> np.sinh(np.pi*1j/2)
1j
>>> np.sinh(np.pi*1j) # (exact value is 0)
1.2246063538223773e-016j
>>> # Discrepancy due to vagaries of floating point arithmetic.
```

```
>>> # Example of providing the optional output parameter
>>> out1 = np.array([0], dtype='d')
>>> out2 = np.sinh([0.1], out1)
>>> out2 is out1
True
```

```
>>> # Example of ValueError due to provision of shape mis-matched `out`
>>> np.sinh(np.zeros((3,3)),np.zeros((2,2)))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: operands could not be broadcast together with shapes (3,3) (2,2)
```

`numpy.`**`cosh`**(*x*, */*, *out=None*, ***, *where=True*, *casting='same_kind'*, *order='K'*, *dtype=None*, *subok=True*[, *signature*, *extobj*]) **`= <ufunc 'cosh'>`**

Hyperbolic cosine, element-wise.

Equivalent to `1/2 * (np.exp(x) + np.exp(-x))` and `np.cos(1j*x)`.

**Parameters**

**x**

[array_like] Input array.

**out**

[ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

**where**

[array_like, optional] This condition is broadcast over the input. At locations where the condition is True, the *out* array will be set to the ufunc result. Elsewhere, the *out* array will retain its original value. Note that if an uninitialized *out* array is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.

****kwargs**

For other keyword-only arguments, see the *ufunc docs*.

**Returns**

**out**

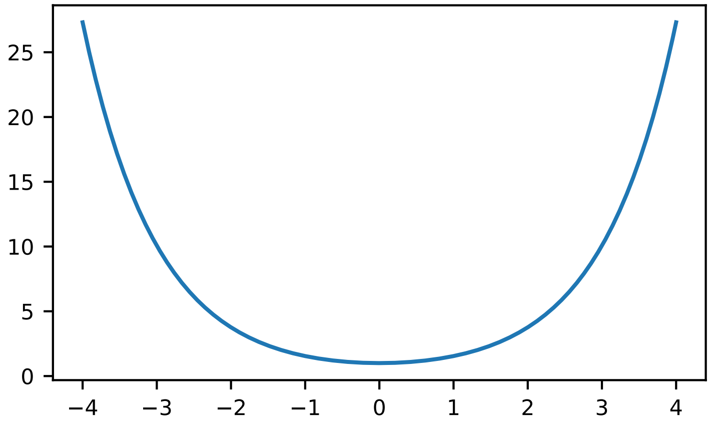
[ndarray or scalar] Output array of same shape as *x*. This is a scalar if *x* is a scalar.

## Examples

```
>>> np.cosh(0)
1.0
```

The hyperbolic cosine describes the shape of a hanging cable:

```
>>> import matplotlib.pyplot as plt
>>> x = np.linspace(-4, 4, 1000)
>>> plt.plot(x, np.cosh(x))
>>> plt.show()
```



`numpy.tanh`(*x*, */*, *out=None*, ***, *where=True*, *casting='same_kind'*, *order='K'*, *dtype=None*, *subok=True*[, *signature*, *extobj* ]) = <ufunc 'tanh'>

Compute hyperbolic tangent element-wise.

Equivalent to `np.sinh(x)/np.cosh(x)` or `-1j * np.tan(1j*x)`.

**Parameters**

**x**

[array_like] Input array.

**out**

[ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

**where**

[array_like, optional] This condition is broadcast over the input. At locations where the condition is True, the *out* array will be set to the ufunc result. Elsewhere, the *out* array will retain its original value. Note that if an uninitialized *out* array is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.

****kwargs**

For other keyword-only arguments, see the *ufunc docs*.

**Returns**

**y**

[ndarray] The corresponding hyperbolic tangent values. This is a scalar if *x* is a scalar.

## Notes

If *out* is provided, the function writes the result into it, and returns a reference to *out*. (See Examples)

## References

[1], [2]

## Examples

```
>>> np.tanh((0, np.pi*1j, np.pi*1j/2))
array([ 0. +0.00000000e+00j,  0. -1.22460635e-16j,  0. +1.63317787e+16j])
```

```
>>> # Example of providing the optional output parameter illustrating
>>> # that what is returned is a reference to said parameter
>>> out1 = np.array([0], dtype='d')
>>> out2 = np.tanh([0.1], out1)
>>> out2 is out1
True
```

```
>>> # Example of ValueError due to provision of shape mis-matched `out`
>>> np.tanh(np.zeros((3,3)),np.zeros((2,2)))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: operands could not be broadcast together with shapes (3,3) (2,2)
```

`numpy.`**`arcsinh`**(*x*, */*, *out=None*, ***, *where=True*, *casting='same_kind'*, *order='K'*, *dtype=None*, *subok=True*[, *signature*, *extobj* ]) **`= <ufunc 'arcsinh'>`**

Inverse hyperbolic sine element-wise.

**Parameters**

**x**

[array_like] Input array.

**out**

[ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

**where**

[array_like, optional] This condition is broadcast over the input. At locations where the condition is True, the *out* array will be set to the ufunc result. Elsewhere, the *out* array will retain its original value. Note that if an uninitialized *out* array is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.

****kwargs**

For other keyword-only arguments, see the *ufunc docs*.

**Returns**

**out**

[ndarray or scalar] Array of the same shape as *x*. This is a scalar if *x* is a scalar.

## Notes

`arcsinh` is a multivalued function: for each *x* there are infinitely many numbers *z* such that *sinh(z) = x*. The convention is to return the *z* whose imaginary part lies in *[-pi/2, pi/2]*.

For real-valued input data types, `arcsinh` always returns real output. For each value that cannot be expressed as a real number or infinity, it returns `nan` and sets the *invalid* floating point error flag.

For complex-valued input, `arccos` is a complex analytical function that has branch cuts *[1j, infj]* and *[-1j, -infj]* and is continuous from the right on the former and from the left on the latter.

The inverse hyperbolic sine is also known as *asinh* or `sinh^-1`.

## References

[1], [2]

## Examples

```
>>> np.arcsinh(np.array([np.e, 10.0]))
array([ 1.72538256,  2.99822295])
```

`numpy.`**`arccosh`**(*x*, */*, *out=None*, ***, *where=True*, *casting='same_kind'*, *order='K'*, *dtype=None*, *subok=True*[, *signature*, *extobj* ]) **`= <ufunc 'arccosh'>`**

Inverse hyperbolic cosine, element-wise.

**Parameters**

**x**

[array_like] Input array.

**out**

[ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

**where**

[array_like, optional] This condition is broadcast over the input. At locations where the condition is True, the *out* array will be set to the ufunc result. Elsewhere, the *out* array will retain its original value. Note that if an uninitialized *out* array is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.

****kwargs**

For other keyword-only arguments, see the *ufunc docs*.

**Returns**

**arccosh**

[ndarray] Array of the same shape as *x*. This is a scalar if *x* is a scalar.

**See also:**

`cosh`, `arcsinh`, `sinh`, `arctanh`, `tanh`

## Notes

`arccosh` is a multivalued function: for each *x* there are infinitely many numbers *z* such that *cosh(z) = x*. The convention is to return the *z* whose imaginary part lies in *[-pi, pi]* and the real part in `[0, inf]`.

For real-valued input data types, `arccosh` always returns real output. For each value that cannot be expressed as a real number or infinity, it yields `nan` and sets the *invalid* floating point error flag.

For complex-valued input, `arccosh` is a complex analytical function that has a branch cut *[-inf, 1]* and is continuous from above on it.

### References

[1], [2]

### Examples

```
>>> np.arccosh([np.e, 10.0])
array([ 1.65745445,  2.99322285])
>>> np.arccosh(1)
0.0
```

`numpy.`**`arctanh`**(*x*, */*, *out=None*, ***, *where=True*, *casting='same_kind'*, *order='K'*, *dtype=None*, *subok=True*[, *signature*, *extobj* ]) **`= <ufunc 'arctanh'>`**

Inverse hyperbolic tangent element-wise.

**Parameters**

**x**

[array_like] Input array.

**out**

[ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

**where**

[array_like, optional] This condition is broadcast over the input. At locations where the condition is True, the *out* array will be set to the ufunc result. Elsewhere, the *out* array will retain its original value. Note that if an uninitialized *out* array is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.

****kwargs**

For other keyword-only arguments, see the *ufunc docs*.

**Returns**

**out**

[ndarray or scalar] Array of the same shape as *x*. This is a scalar if *x* is a scalar.

**See also:**

`emath.arctanh`

### Notes

`arctanh` is a multivalued function: for each *x* there are infinitely many numbers *z* such that *tanh(z) = x*. The convention is to return the *z* whose imaginary part lies in *[-pi/2, pi/2]*.

For real-valued input data types, `arctanh` always returns real output. For each value that cannot be expressed as a real number or infinity, it yields `nan` and sets the *invalid* floating point error flag.

For complex-valued input, `arctanh` is a complex analytical function that has branch cuts *[-1, -inf]* and *[1, inf]* and is continuous from above on the former and from below on the latter.

The inverse hyperbolic tangent is also known as *atanh* or `tanh^-1`.

### References

[1], [2]

### Examples

```
>>> np.arctanh([0, -0.5])
array([ 0.        , -0.54930614])
```

## 4.19.3 Rounding

| | |
|---|---|
| `around`(a[, decimals, out]) | Evenly round to the given number of decimals. |
| `round_`(a[, decimals, out]) | Round an array to the given number of decimals. |
| `rint`(x, /[, out, where, casting, order, …]) | Round elements of the array to the nearest integer. |
| `fix`(x[, out]) | Round to nearest integer towards zero. |
| `floor`(x, /[, out, where, casting, order, …]) | Return the floor of the input, element-wise. |
| `ceil`(x, /[, out, where, casting, order, …]) | Return the ceiling of the input, element-wise. |
| `trunc`(x, /[, out, where, casting, order, …]) | Return the truncated value of the input, element-wise. |

`numpy.`**`around`**`(`*a*, *decimals=0*, *out=None*`)`

Evenly round to the given number of decimals.

**Parameters**

**a**

[array_like] Input data.

**decimals**

[int, optional] Number of decimal places to round to (default: 0). If decimals is negative, it specifies the number of positions to the left of the decimal point.

**out**

[ndarray, optional] Alternative output array in which to place the result. It must have the same shape as the expected output, but the type of the output values will be cast if necessary. See *ufuncs-output-type* for more details.

**Returns**

**rounded_array**

[ndarray] An array of the same type as *a*, containing the rounded values. Unless *out* was specified, a new array is created. A reference to the result is returned.

The real and imaginary parts of complex numbers are rounded separately. The result of rounding a float is a float.

**See also:**

***ndarray.round***

equivalent method

*ceil*, *fix*, *floor*, *rint*, *trunc*

## Notes

For values exactly halfway between rounded decimal values, NumPy rounds to the nearest even value. Thus 1.5 and 2.5 round to 2.0, -0.5 and 0.5 round to 0.0, etc.

`np.around` uses a fast but sometimes inexact algorithm to round floating-point datatypes. For positive *decimals* it is equivalent to `np.true_divide(np.rint(a * 10**decimals), 10**decimals)`, which has error due to the inexact representation of decimal fractions in the IEEE floating point standard [1] and errors introduced when scaling by powers of ten. For instance, note the extra "1" in the following:

```
>>> np.round(56294995342131.5, 3)
56294995342131.51
```

If your goal is to print such values with a fixed number of decimals, it is preferable to use numpy's float printing routines to limit the number of printed decimals:

```
>>> np.format_float_positional(56294995342131.5, precision=3)
'56294995342131.5'
```

The float printing routines use an accurate but much more computationally demanding algorithm to compute the number of digits after the decimal point.

Alternatively, Python's builtin `round` function uses a more accurate but slower algorithm for 64-bit floating point values:

```
>>> round(56294995342131.5, 3)
56294995342131.5
>>> np.round(16.055, 2), round(16.055, 2)  # equals 16.0549999999999997
(16.06, 16.05)
```

## References

[1], [2]

### Examples

```
>>> np.around([0.37, 1.64])
array([0.,  2.])
>>> np.around([0.37, 1.64], decimals=1)
array([0.4,  1.6])
>>> np.around([.5, 1.5, 2.5, 3.5, 4.5]) # rounds to nearest even value
array([0.,  2.,  2.,  4.,  4.])
>>> np.around([1,2,3,11], decimals=1) # ndarray of ints is returned
array([ 1,  2,  3, 11])
>>> np.around([1,2,3,11], decimals=-1)
array([ 0,  0,  0, 10])
```

`numpy.`**`round_`**(*a*, *decimals=0*, *out=None*)

Round an array to the given number of decimals.

**See also:**

***around***

equivalent function; see for details.

`numpy.`**`rint`**(*x*, */*, *out=None*, ***, *where=True*, *casting='same_kind'*, *order='K'*, *dtype=None*, *subok=True*[, *signature*, *extobj*]) **`= <ufunc 'rint'>`**

Round elements of the array to the nearest integer.

**Parameters**

**x**

[array_like] Input array.

**out**

[ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

**where**

[array_like, optional] This condition is broadcast over the input. At locations where the condition is True, the *out* array will be set to the ufunc result. Elsewhere, the *out* array will retain its original value. Note that if an uninitialized *out* array is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.

****kwargs**

For other keyword-only arguments, see the *ufunc docs*.

**Returns**

**out**

[ndarray or scalar] Output array is same shape and type as *x*. This is a scalar if *x* is a scalar.

**See also:**

*ceil*, *floor*, *trunc*

### Examples

```
>>> a = np.array([-1.7, -1.5, -0.2, 0.2, 1.5, 1.7, 2.0])
>>> np.rint(a)
array([-2., -2., -0.,  0.,  2.,  2.,  2.])
```

`numpy.`**`fix`**(*x*, *out=None*)

Round to nearest integer towards zero.

Round an array of floats element-wise to nearest integer towards zero. The rounded values are returned as floats.

**Parameters**

**x**

[array_like] An array of floats to be rounded

**out**

[ndarray, optional] A location into which the result is stored. If provided, it must have a shape that the input broadcasts to. If not provided or None, a freshly-allocated array is returned.

**Returns**

**out**

[ndarray of floats] A float array with the same dimensions as the input. If second argument is not supplied then a float array is returned with the rounded values.

If a second argument is supplied the result is stored there. The return value *out* is then a reference to that array.

**See also:**

*`trunc`*, *`floor`*, *`ceil`*

***`around`***

Round to given number of decimals

### Examples

```
>>> np.fix(3.14)
3.0
>>> np.fix(3)
3.0
>>> np.fix([2.1, 2.9, -2.1, -2.9])
array([ 2.,  2., -2., -2.])
```

`numpy.`**`floor`**(*x*, */*, *out=None*, ***, *where=True*, *casting='same_kind'*, *order='K'*, *dtype=None*, *subok=True*[, *signature*, *extobj*]) **`= <ufunc 'floor'>`**

Return the floor of the input, element-wise.

The floor of the scalar *x* is the largest integer *i*, such that *i* <= *x*. It is often denoted as $\lfloor x \rfloor$.

**Parameters**

**x**

[array_like] Input data.

**out**

[ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

**where**

[array_like, optional] This condition is broadcast over the input. At locations where the condition is True, the *out* array will be set to the ufunc result. Elsewhere, the *out* array will retain its original value. Note that if an uninitialized *out* array is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.

****kwargs**

For other keyword-only arguments, see the *ufunc docs*.

**Returns**

**y**

[ndarray or scalar] The floor of each element in *x*. This is a scalar if *x* is a scalar.

**See also:**

`ceil`, `trunc`, `rint`

### Notes

Some spreadsheet programs calculate the "floor-towards-zero", in other words `floor(-2.5) == -2`. NumPy instead uses the definition of `floor` where *floor(-2.5) == -3*.

### Examples

```
>>> a = np.array([-1.7, -1.5, -0.2, 0.2, 1.5, 1.7, 2.0])
>>> np.floor(a)
array([-2., -2., -1.,  0.,  1.,  1.,  2.])
```

`numpy.`**`ceil`**(*x*, */*, *out=None*, ***, *where=True*, *casting='same_kind'*, *order='K'*, *dtype=None*, *subok=True*[, *signature*, *extobj*]) **`= <ufunc 'ceil'>`**

Return the ceiling of the input, element-wise.

The ceil of the scalar *x* is the smallest integer *i*, such that *i* >= *x*. It is often denoted as $\lceil x \rceil$.

**Parameters**

**x**

[array_like] Input data.

**out**

[ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

**where**

[array_like, optional] This condition is broadcast over the input. At locations where the condition is True, the *out* array will be set to the ufunc result. Elsewhere, the *out* array will retain its original value. Note that if an uninitialized *out* array is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.

****kwargs**

For other keyword-only arguments, see the *ufunc docs*.

**Returns**

**y**

[ndarray or scalar] The ceiling of each element in *x*, with `float` dtype. This is a scalar if *x* is a scalar.

**See also:**

`floor`, `trunc`, `rint`

### Examples

```
>>> a = np.array([-1.7, -1.5, -0.2, 0.2, 1.5, 1.7, 2.0])
>>> np.ceil(a)
array([-1., -1., -0.,  1.,  2.,  2.,  2.])
```

`numpy.trunc`(*x*, */*, *out=None*, ***, *where=True*, *casting='same_kind'*, *order='K'*, *dtype=None*, *subok=True*[, *signature*, *extobj*]) **= <ufunc 'trunc'>**

Return the truncated value of the input, element-wise.

The truncated value of the scalar *x* is the nearest integer *i* which is closer to zero than *x* is. In short, the fractional part of the signed number *x* is discarded.

**Parameters**

**x**

[array_like] Input data.

**out**

[ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

**where**

[array_like, optional] This condition is broadcast over the input. At locations where the condition is True, the *out* array will be set to the ufunc result. Elsewhere, the *out* array will retain its original value. Note that if an uninitialized *out* array is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.

****kwargs**

For other keyword-only arguments, see the *ufunc docs*.

**Returns**

**y**

[ndarray or scalar] The truncated value of each element in *x*. This is a scalar if *x* is a scalar.

**See also:**

`ceil`, `floor`, `rint`

### Notes

New in version 1.3.0.

### Examples

```
>>> a = np.array([-1.7, -1.5, -0.2, 0.2, 1.5, 1.7, 2.0])
>>> np.trunc(a)
array([-1., -1., -0.,  0.,  1.,  1.,  2.])
```

## 4.19.4 Sums, products, differences

| | |
|---|---|
| `prod`(a[, axis, dtype, out, keepdims, …]) | Return the product of array elements over a given axis. |
| `sum`(a[, axis, dtype, out, keepdims, …]) | Sum of array elements over a given axis. |
| `nanprod`(a[, axis, dtype, out, keepdims]) | Return the product of array elements over a given axis treating Not a Numbers (NaNs) as ones. |
| `nansum`(a[, axis, dtype, out, keepdims]) | Return the sum of array elements over a given axis treating Not a Numbers (NaNs) as zero. |
| `cumprod`(a[, axis, dtype, out]) | Return the cumulative product of elements along a given axis. |
| `cumsum`(a[, axis, dtype, out]) | Return the cumulative sum of the elements along a given axis. |
| `nancumprod`(a[, axis, dtype, out]) | Return the cumulative product of array elements over a given axis treating Not a Numbers (NaNs) as one. |
| `nancumsum`(a[, axis, dtype, out]) | Return the cumulative sum of array elements over a given axis treating Not a Numbers (NaNs) as zero. |
| `diff`(a[, n, axis, prepend, append]) | Calculate the n-th discrete difference along the given axis. |
| `ediff1d`(ary[, to_end, to_begin]) | The differences between consecutive elements of an array. |
| `gradient`(f, *varargs[, axis, edge_order]) | Return the gradient of an N-dimensional array. |
| `cross`(a, b[, axisa, axisb, axisc, axis]) | Return the cross product of two (arrays of) vectors. |
| `trapz`(y[, x, dx, axis]) | Integrate along the given axis using the composite trapezoidal rule. |

`numpy.`**`prod`**(*a*, *axis=None*, *dtype=None*, *out=None*, *keepdims=<no value>*, *initial=<no value>*, *where=<no value>*)

Return the product of array elements over a given axis.

**Parameters**

**a**

[array_like] Input data.

**axis**

[None or int or tuple of ints, optional] Axis or axes along which a product is performed. The default, axis=None, will calculate the product of all the elements in the input array. If axis is negative it counts from the last to the first axis.

New in version 1.7.0.

If axis is a tuple of ints, a product is performed on all of the axes specified in the tuple instead of a single axis or all the axes as before.

**dtype**

[dtype, optional] The type of the returned array, as well as of the accumulator in which the elements are multiplied. The dtype of *a* is used by default unless *a* has an integer dtype of less precision than the default platform integer. In that case, if *a* is signed then the platform integer is used while if *a* is unsigned then an unsigned integer of the same precision as the platform integer is used.

**out**

[ndarray, optional] Alternative output array in which to place the result. It must have the same shape as the expected output, but the type of the output values will be cast if necessary.

**keepdims**

[bool, optional] If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the input array.

If the default value is passed, then *keepdims* will not be passed through to the `prod` method of sub-classes of `ndarray`, however any non-default value will be. If the sub-class' method does not implement *keepdims* any exceptions will be raised.

**initial**

[scalar, optional] The starting value for this product. See `reduce` for details.

New in version 1.15.0.

**where**

[array_like of bool, optional] Elements to include in the product. See `reduce` for details.

New in version 1.17.0.

**Returns**

**product_along_axis**

[ndarray, see `dtype` parameter above.] An array shaped as *a* but with the specified axis removed. Returns a reference to *out* if specified.

**See also:**

**`ndarray.prod`**

equivalent method

`ufuncs-output-type`

### Notes

Arithmetic is modular when using integer types, and no error is raised on overflow. That means that, on a 32-bit platform:

```
>>> x = np.array([536870910, 536870910, 536870910, 536870910])
>>> np.prod(x)
16 # may vary
```

The product of an empty array is the neutral element 1:

```
>>> np.prod([])
1.0
```

### Examples

By default, calculate the product of all elements:

```
>>> np.prod([1.,2.])
2.0
```

Even when the input array is two-dimensional:

```
>>> np.prod([[1.,2.],[3.,4.]])
24.0
```

But we can also specify the axis over which to multiply:

```
>>> np.prod([[1.,2.],[3.,4.]], axis=1)
array([  2.,  12.])
```

Or select specific elements to include:

```
>>> np.prod([1., np.nan, 3.], where=[True, False, True])
3.0
```

If the type of *x* is unsigned, then the output type is the unsigned platform integer:

```
>>> x = np.array([1, 2, 3], dtype=np.uint8)
>>> np.prod(x).dtype == np.uint
True
```

If *x* is of a signed integer type, then the output type is the default platform integer:

```
>>> x = np.array([1, 2, 3], dtype=np.int8)
>>> np.prod(x).dtype == int
True
```

You can also start the product with a value other than one:

```
>>> np.prod([1, 2], initial=5)
10
```

`numpy.`**`sum`**(*a*, *axis=None*, *dtype=None*, *out=None*, *keepdims=<no value>*, *initial=<no value>*, *where=<no value>*)

Sum of array elements over a given axis.

**Parameters**

**a**

[array_like] Elements to sum.

**axis**

[None or int or tuple of ints, optional] Axis or axes along which a sum is performed. The default, axis=None, will sum all of the elements of the input array. If axis is negative it counts from the last to the first axis.

New in version 1.7.0.

If axis is a tuple of ints, a sum is performed on all of the axes specified in the tuple instead of a single axis or all the axes as before.

**dtype**

[dtype, optional] The type of the returned array and of the accumulator in which the elements are summed. The dtype of *a* is used by default unless *a* has an integer dtype of less precision than the default platform integer. In that case, if *a* is signed then the platform integer is used while if *a* is unsigned then an unsigned integer of the same precision as the platform integer is used.

**out**

[ndarray, optional] Alternative output array in which to place the result. It must have the same shape as the expected output, but the type of the output values will be cast if necessary.

**keepdims**

[bool, optional] If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the input array.

If the default value is passed, then *keepdims* will not be passed through to the `sum` method of sub-classes of `ndarray`, however any non-default value will be. If the sub-class' method does not implement *keepdims* any exceptions will be raised.

**initial**

[scalar, optional] Starting value for the sum. See `reduce` for details.

New in version 1.15.0.

**where**

[array_like of bool, optional] Elements to include in the sum. See `reduce` for details.

New in version 1.17.0.

**Returns**

**sum_along_axis**

[ndarray] An array with the same shape as *a*, with the specified axis removed. If *a* is a 0-d array, or if *axis* is None, a scalar is returned. If an output array is specified, a reference to *out* is returned.

**See also:**

***`ndarray.sum`***

Equivalent method.

**`add.reduce`**

Equivalent functionality of *`add`*.

***`cumsum`***

Cumulative sum of array elements.

***`trapz`***

Integration of array values using the composite trapezoidal rule.

*`mean`*, *`average`*

## Notes

Arithmetic is modular when using integer types, and no error is raised on overflow.

The sum of an empty array is the neutral element 0:

```
>>> np.sum([])
0.0
```

For floating point numbers the numerical precision of sum (and `np.add.reduce`) is in general limited by directly adding each number individually to the result causing rounding errors in every step. However, often numpy will use a numerically better approach (partial pairwise summation) leading to improved precision in many use-cases. This improved precision is always provided when no `axis` is given. When `axis` is given, it will depend on which axis is summed. Technically, to provide the best speed possible, the improved precision is only used when the summation is along the fast axis in memory. Note that the exact precision may vary depending on other parameters. In contrast to NumPy, Python's `math.fsum` function uses a slower but more precise approach to summation. Especially when summing a large number of lower precision floating point numbers, such as `float32`, numerical errors can become significant. In such cases it can be advisable to use *dtype="float64"* to use a higher precision for the output.

## Examples

```
>>> np.sum([0.5, 1.5])
2.0
>>> np.sum([0.5, 0.7, 0.2, 1.5], dtype=np.int32)
1
>>> np.sum([[0, 1], [0, 5]])
6
>>> np.sum([[0, 1], [0, 5]], axis=0)
array([0, 6])
>>> np.sum([[0, 1], [0, 5]], axis=1)
array([1, 5])
>>> np.sum([[0, 1], [np.nan, 5]], where=[False, True], axis=1)
array([1., 5.])
```

If the accumulator is too small, overflow occurs:

```
>>> np.ones(128, dtype=np.int8).sum(dtype=np.int8)
-128
```

You can also start the sum with a value other than zero:

```
>>> np.sum([10], initial=5)
15
```

`numpy.`**`nanprod`**(*a*, *axis=None*, *dtype=None*, *out=None*, *keepdims=<no value>*)

Return the product of array elements over a given axis treating Not a Numbers (NaNs) as ones.

One is returned for slices that are all-NaN or empty.

New in version 1.10.0.

**Parameters**

**a**

[array_like] Array containing numbers whose product is desired. If *a* is not an array, a conversion is attempted.

**axis**

[{int, tuple of int, None}, optional] Axis or axes along which the product is computed. The default is to compute the product of the flattened array.

**dtype**

[data-type, optional] The type of the returned array and of the accumulator in which the elements are summed. By default, the dtype of *a* is used. An exception is when *a* has an integer type with less precision than the platform (u)intp. In that case, the default will be either (u)int32 or (u)int64 depending on whether the platform is 32 or 64 bits. For inexact inputs, dtype must be inexact.

**out**

[ndarray, optional] Alternate output array in which to place the result. The default is `None`. If provided, it must have the same shape as the expected output, but the type will be cast if necessary. See *ufuncs-output-type* for more details. The casting of NaN to integer can yield unexpected results.

**keepdims**

[bool, optional] If True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the original *arr*.

**Returns**

**nanprod**

[ndarray] A new array holding the result is returned unless *out* is specified, in which case it is returned.

**See also:**

***`numpy.prod`***

Product across array propagating NaNs.

***`isnan`***

Show which elements are NaN.

## Examples

```
>>> np.nanprod(1)
1
>>> np.nanprod([1])
1
>>> np.nanprod([1, np.nan])
1.0
>>> a = np.array([[1, 2], [3, np.nan]])
>>> np.nanprod(a)
6.0
>>> np.nanprod(a, axis=0)
array([3., 2.])
```

`numpy.`**`nansum`**(*a*, *axis=None*, *dtype=None*, *out=None*, *keepdims=<no value>*)

Return the sum of array elements over a given axis treating Not a Numbers (NaNs) as zero.

In NumPy versions <= 1.9.0 Nan is returned for slices that are all-NaN or empty. In later versions zero is returned.

**Parameters**

**a**

[array_like] Array containing numbers whose sum is desired. If *a* is not an array, a conversion is attempted.

**axis**

[{int, tuple of int, None}, optional] Axis or axes along which the sum is computed. The default is to compute the sum of the flattened array.

**dtype**

[data-type, optional] The type of the returned array and of the accumulator in which the elements are summed. By default, the dtype of *a* is used. An exception is when *a* has an integer type with less precision than the platform (u)intp. In that case, the default will be either (u)int32 or (u)int64 depending on whether the platform is 32 or 64 bits. For inexact inputs, dtype must be inexact.

New in version 1.8.0.

**out**

[ndarray, optional] Alternate output array in which to place the result. The default is `None`. If provided, it must have the same shape as the expected output, but the type will be cast if necessary. See *ufuncs-output-type* for more details. The casting of NaN to integer can yield unexpected results.

New in version 1.8.0.

**keepdims**

[bool, optional] If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the original *a*.

If the value is anything but the default, then *keepdims* will be passed through to the `mean` or `sum` methods of sub-classes of `ndarray`. If the sub-classes methods does not implement *keepdims* any exceptions will be raised.

New in version 1.8.0.

**Returns**

**nansum**

[ndarray.] A new array holding the result is returned unless *out* is specified, in which it is returned. The result has the same size as *a*, and the same shape as *a* if *axis* is not None or *a* is a 1-d array.

**See also:**

**`numpy.sum`**

Sum across array propagating NaNs.

**`isnan`**

Show which elements are NaN.

**`isfinite`**

Show which elements are not NaN or +/-inf.

## Notes

If both positive and negative infinity are present, the sum will be Not A Number (NaN).

## Examples

```
>>> np.nansum(1)
1
>>> np.nansum([1])
1
>>> np.nansum([1, np.nan])
1.0
>>> a = np.array([[1, 1], [1, np.nan]])
>>> np.nansum(a)
3.0
>>> np.nansum(a, axis=0)
array([2.,  1.])
>>> np.nansum([1, np.nan, np.inf])
inf
>>> np.nansum([1, np.nan, np.NINF])
-inf
>>> from numpy.testing import suppress_warnings
>>> with suppress_warnings() as sup:
...     sup.filter(RuntimeWarning)
...     np.nansum([1, np.nan, np.inf, -np.inf]) # both +/- infinity present
nan
```

`numpy.`**`cumprod`**(*a*, *axis=None*, *dtype=None*, *out=None*)

Return the cumulative product of elements along a given axis.

**Parameters**

**a**

[array_like] Input array.

**axis**

[int, optional] Axis along which the cumulative product is computed. By default the input is flattened.

**dtype**

[dtype, optional] Type of the returned array, as well as of the accumulator in which the elements are multiplied. If *dtype* is not specified, it defaults to the dtype of *a*, unless *a* has an integer dtype with a precision less than that of the default platform integer. In that case, the default platform integer is used instead.

**out**

[ndarray, optional] Alternative output array in which to place the result. It must have the same shape and buffer length as the expected output but the type of the resulting values will be cast if necessary.

**Returns**

**cumprod**

[ndarray] A new array holding the result is returned unless *out* is specified, in which case a reference to out is returned.

**See also:**

`ufuncs-output-type`

## Notes

Arithmetic is modular when using integer types, and no error is raised on overflow.

## Examples

```
>>> a = np.array([1,2,3])
>>> np.cumprod(a) # intermediate results 1, 1*2
...               # total product 1*2*3 = 6
array([1, 2, 6])
>>> a = np.array([[1, 2, 3], [4, 5, 6]])
>>> np.cumprod(a, dtype=float) # specify type of output
array([   1.,    2.,    6.,   24.,  120.,  720.])
```

The cumulative product for each column (i.e., over the rows) of *a*:

```
>>> np.cumprod(a, axis=0)
array([[ 1,  2,  3],
       [ 4, 10, 18]])
```

The cumulative product for each row (i.e. over the columns) of *a*:

```
>>> np.cumprod(a,axis=1)
array([[  1,   2,   6],
       [  4,  20, 120]])
```

`numpy.`**`cumsum`**(*a*, *axis=None*, *dtype=None*, *out=None*)

Return the cumulative sum of the elements along a given axis.

**Parameters**

**a**

[array_like] Input array.

**axis**

[int, optional] Axis along which the cumulative sum is computed. The default (None) is to compute the cumsum over the flattened array.

**dtype**

[dtype, optional] Type of the returned array and of the accumulator in which the elements are summed. If `dtype` is not specified, it defaults to the dtype of *a*, unless *a* has an integer dtype with a precision less than that of the default platform integer. In that case, the default platform integer is used.

**out**

[ndarray, optional] Alternative output array in which to place the result. It must have the same shape and buffer length as the expected output but the type will be cast if necessary. See *ufuncs-output-type* for more details.

**Returns**

**cumsum_along_axis**

[ndarray.] A new array holding the result is returned unless *out* is specified, in which case a reference to *out* is returned. The result has the same size as *a*, and the same shape as *a* if *axis* is not None or *a* is a 1-d array.

**See also:**

**`sum`**

Sum array elements.

**`trapz`**

Integration of array values using the composite trapezoidal rule.

**`diff`**

Calculate the n-th discrete difference along given axis.

## Notes

Arithmetic is modular when using integer types, and no error is raised on overflow.

### Examples

```
>>> a = np.array([[1,2,3], [4,5,6]])
>>> a
array([[1, 2, 3],
       [4, 5, 6]])
>>> np.cumsum(a)
array([ 1,  3,  6, 10, 15, 21])
>>> np.cumsum(a, dtype=float)     # specifies type of output value(s)
array([  1.,   3.,   6.,  10.,  15.,  21.])
```

```
>>> np.cumsum(a,axis=0)      # sum over rows for each of the 3 columns
array([[1, 2, 3],
       [5, 7, 9]])
>>> np.cumsum(a,axis=1)      # sum over columns for each of the 2 rows
array([[ 1,  3,  6],
       [ 4,  9, 15]])
```

`numpy.`**`nancumprod`**(*a*, *axis=None*, *dtype=None*, *out=None*)

Return the cumulative product of array elements over a given axis treating Not a Numbers (NaNs) as one. The cumulative product does not change when NaNs are encountered and leading NaNs are replaced by ones.

Ones are returned for slices that are all-NaN or empty.

New in version 1.12.0.

**Parameters**

**a**

[array_like] Input array.

**axis**

[int, optional] Axis along which the cumulative product is computed. By default the input is flattened.

**dtype**

[dtype, optional] Type of the returned array, as well as of the accumulator in which the elements are multiplied. If *dtype* is not specified, it defaults to the dtype of *a*, unless *a* has an integer dtype with a precision less than that of the default platform integer. In that case, the default platform integer is used instead.

**out**

[ndarray, optional] Alternative output array in which to place the result. It must have the same shape and buffer length as the expected output but the type of the resulting values will be cast if necessary.

**Returns**

**nancumprod**

[ndarray] A new array holding the result is returned unless *out* is specified, in which case it is returned.

**See also:**

*`numpy.cumprod`*

Cumulative product across array propagating NaNs.

*`isnan`*

Show which elements are NaN.

### Examples

```
>>> np.nancumprod(1)
array([1])
>>> np.nancumprod([1])
array([1])
>>> np.nancumprod([1, np.nan])
array([1.,  1.])
>>> a = np.array([[1, 2], [3, np.nan]])
>>> np.nancumprod(a)
array([1.,  2.,  6.,  6.])
>>> np.nancumprod(a, axis=0)
array([[1.,  2.],
       [3.,  2.]])
>>> np.nancumprod(a, axis=1)
array([[1.,  2.],
       [3.,  3.]])
```

`numpy.`**`nancumsum`**(*a*, *axis=None*, *dtype=None*, *out=None*)

Return the cumulative sum of array elements over a given axis treating Not a Numbers (NaNs) as zero. The cumulative sum does not change when NaNs are encountered and leading NaNs are replaced by zeros.

Zeros are returned for slices that are all-NaN or empty.

New in version 1.12.0.

**Parameters**

**a**

[array_like] Input array.

**axis**

[int, optional] Axis along which the cumulative sum is computed. The default (None) is to compute the cumsum over the flattened array.

**dtype**

[dtype, optional] Type of the returned array and of the accumulator in which the elements are summed. If *`dtype`* is not specified, it defaults to the dtype of *a*, unless *a* has an integer dtype with a precision less than that of the default platform integer. In that case, the default platform integer is used.

**out**

[ndarray, optional] Alternative output array in which to place the result. It must have the same shape and buffer length as the expected output but the type will be cast if necessary. See *ufuncs-output-type* for more details.

**Returns**

**nancumsum**

[ndarray.] A new array holding the result is returned unless *out* is specified, in which it is returned. The result has the same size as *a*, and the same shape as *a* if *axis* is not None or *a* is a 1-d array.

**See also:**

[`numpy.cumsum`]

Cumulative sum across array propagating NaNs.

[`isnan`]

Show which elements are NaN.

## Examples

```
>>> np.nancumsum(1)
array([1])
>>> np.nancumsum([1])
array([1])
>>> np.nancumsum([1, np.nan])
array([1.,  1.])
>>> a = np.array([[1, 2], [3, np.nan]])
>>> np.nancumsum(a)
array([1.,  3.,  6.,  6.])
>>> np.nancumsum(a, axis=0)
array([[1.,  2.],
       [4.,  2.]])
>>> np.nancumsum(a, axis=1)
array([[1.,  3.],
       [3.,  3.]])
```

`numpy.diff`(*a*, *n=1*, *axis=-1*, *prepend=<no value>*, *append=<no value>*)

Calculate the n-th discrete difference along the given axis.

The first difference is given by `out[i] = a[i+1] - a[i]` along the given axis, higher differences are calculated by using `diff` recursively.

**Parameters**

**a**

[array_like] Input array

**n**

[int, optional] The number of times values are differenced. If zero, the input is returned as-is.

**axis**

[int, optional] The axis along which the difference is taken, default is the last axis.

**prepend, append**

[array_like, optional] Values to prepend or append to *a* along axis prior to performing the difference. Scalar values are expanded to arrays with length 1 in the direction of axis and the shape of the input array in along all other axes. Otherwise the dimension and shape must match *a* except along axis.

New in version 1.16.0.

**Returns**

**diff**

[ndarray] The n-th differences. The shape of the output is the same as *a* except along *axis* where the dimension is smaller by *n*. The type of the output is the same as the type of the difference between any two elements of *a*. This is the same as the type of *a* in most cases. A notable exception is `datetime64`, which results in a `timedelta64` output array.

**See also:**

`gradient`, `ediff1d`, `cumsum`

## Notes

Type is preserved for boolean arrays, so the result will contain *False* when consecutive elements are the same and *True* when they differ.

For unsigned integer arrays, the results will also be unsigned. This should not be surprising, as the result is consistent with calculating the difference directly:

```
>>> u8_arr = np.array([1, 0], dtype=np.uint8)
>>> np.diff(u8_arr)
array([255], dtype=uint8)
>>> u8_arr[1,...] - u8_arr[0,...]
255
```

If this is not desirable, then the array should be cast to a larger integer type first:

```
>>> i16_arr = u8_arr.astype(np.int16)
>>> np.diff(i16_arr)
array([-1], dtype=int16)
```

## Examples

```
>>> x = np.array([1, 2, 4, 7, 0])
>>> np.diff(x)
array([ 1,  2,  3, -7])
>>> np.diff(x, n=2)
array([  1,   1, -10])
```

```
>>> x = np.array([[1, 3, 6, 10], [0, 5, 6, 8]])
>>> np.diff(x)
array([[2, 3, 4],
       [5, 1, 2]])
>>> np.diff(x, axis=0)
array([[-1,  2,  0, -2]])
```

```
>>> x = np.arange('1066-10-13', '1066-10-16', dtype=np.datetime64)
>>> np.diff(x)
array([1, 1], dtype='timedelta64[D]')
```

`numpy.`**`ediff1d`**(*ary*, *to_end=None*, *to_begin=None*)

The differences between consecutive elements of an array.

**Parameters**

**ary**

[array_like] If necessary, will be flattened before the differences are taken.

**to_end**

[array_like, optional] Number(s) to append at the end of the returned differences.

**to_begin**

[array_like, optional] Number(s) to prepend at the beginning of the returned differences.

**Returns**

**ediff1d**

[ndarray] The differences. Loosely, this is `ary.flat[1:] - ary.flat[:-1]`.

**See also:**

*`diff`*, *`gradient`*

## Notes

When applied to masked arrays, this function drops the mask information if the *to_begin* and/or *to_end* parameters are used.

## Examples

```
>>> x = np.array([1, 2, 4, 7, 0])
>>> np.ediff1d(x)
array([ 1,  2,  3, -7])
```

```
>>> np.ediff1d(x, to_begin=-99, to_end=np.array([88, 99]))
array([-99,   1,   2, ...,  -7,  88,  99])
```

The returned array is always 1D.

```
>>> y = [[1, 2, 4], [1, 6, 24]]
>>> np.ediff1d(y)
array([ 1,  2, -3,  5, 18])
```

`numpy.`**`gradient`**(*f*, **varargs*, *axis=None*, *edge_order=1*)

Return the gradient of an N-dimensional array.

The gradient is computed using second order accurate central differences in the interior points and either first or second order accurate one-sides (forward or backwards) differences at the boundaries. The returned gradient hence has the same shape as the input array.

**Parameters**

**f**

[array_like] An N-dimensional array containing samples of a scalar function.

**varargs**

[list of scalar or array, optional] Spacing between f values. Default unitary spacing for all dimensions. Spacing can be specified using:

1. single scalar to specify a sample distance for all dimensions.
2. N scalars to specify a constant sample distance for each dimension. i.e. *dx*, *dy*, *dz*, …
3. N arrays to specify the coordinates of the values along each dimension of F. The length of the array must match the size of the corresponding dimension
4. Any combination of N scalars/arrays with the meaning of 2. and 3.

If *axis* is given, the number of varargs must equal the number of axes. Default: 1.

**edge_order**

[$\{1, 2\}$, optional] Gradient is calculated using N-th order accurate differences at the boundaries. Default: 1.

New in version 1.9.1.

**axis**

[None or int or tuple of ints, optional] Gradient is calculated only along the given axis or axes The default (axis = None) is to calculate the gradient for all the axes of the input array. axis may be negative, in which case it counts from the last to the first axis.

New in version 1.11.0.

**Returns**

**gradient**

[ndarray or list of ndarray] A set of ndarrays (or a single ndarray if there is only one dimension) corresponding to the derivatives of f with respect to each dimension. Each derivative has the same shape as f.

## Notes

Assuming that $f \in C^3$ (i.e., $f$ has at least 3 continuous derivatives) and let $h_*$ be a non-homogeneous stepsize, we minimize the "consistency error" $\eta_i$ between the true gradient and its estimate from a linear combination of the neighboring grid-points:

$$\eta_i = f_i^{(1)} - \left[\alpha f(x_i) + \beta f(x_i + h_d) + \gamma f(x_i - h_s)\right]$$

By substituting $f(x_i + h_d)$ and $f(x_i - h_s)$ with their Taylor series expansion, this translates into solving the following the linear system:

$$\left\{\begin{array}{r}\alpha + \beta + \gamma = 0 \\ \beta h_d - \gamma h_s = 1 \\ \beta h_d^2 + \gamma h_s^2 = 0\end{array}\right.$$

The resulting approximation of $f_i^{(1)}$ is the following:

$$\hat f_i^{(1)} = \frac{h_s^2 f(x_i + h_d) + \left(h_d^2 - h_s^2\right) f(x_i) - h_d^2 f(x_i - h_s)}{h_s h_d (h_d + h_s)} + \mathcal{O}\left(\frac{h_d h_s^2 + h_s h_d^2}{h_d + h_s}\right)$$

It is worth noting that if $h_s = h_d$ (i.e., data are evenly spaced) we find the standard second order approximation:

$$\hat{f}_i^{(1)} = \frac{f(x_{i+1}) - f(x_{i-1})}{2h} + \mathcal{O}(h^2)$$

With a similar procedure the forward/backward approximations used for boundaries can be derived.

### References

[1], [2], [3]

### Examples

```
>>> f = np.array([1, 2, 4, 7, 11, 16], dtype=float)
>>> np.gradient(f)
array([1. , 1.5, 2.5, 3.5, 4.5, 5. ])
>>> np.gradient(f, 2)
array([0.5 ,  0.75,  1.25,  1.75,  2.25,  2.5 ])
```

Spacing can be also specified with an array that represents the coordinates of the values F along the dimensions. For instance a uniform spacing:

```
>>> x = np.arange(f.size)
>>> np.gradient(f, x)
array([1. ,  1.5,  2.5,  3.5,  4.5,  5. ])
```

Or a non uniform one:

```
>>> x = np.array([0., 1., 1.5, 3.5, 4., 6.], dtype=float)
>>> np.gradient(f, x)
array([1. ,  3. ,  3.5,  6.7,  6.9,  2.5])
```

For two dimensional arrays, the return will be two arrays ordered by axis. In this example the first array stands for the gradient in rows and the second one in columns direction:

```
>>> np.gradient(np.array([[1, 2, 6], [3, 4, 5]], dtype=float))
[array([[ 2.,  2., -1.],
       [ 2.,  2., -1.]]), array([[1. , 2.5, 4. ],
       [1. , 1. , 1. ]])]
```

In this example the spacing is also specified: uniform for axis=0 and non uniform for axis=1

```
>>> dx = 2.
>>> y = [1., 1.5, 3.5]
>>> np.gradient(np.array([[1, 2, 6], [3, 4, 5]], dtype=float), dx, y)
[array([[ 1. ,  1. , -0.5],
       [ 1. ,  1. , -0.5]]), array([[2. , 2. , 2. ],
       [2. , 1.7, 0.5]])]
```

It is possible to specify how boundaries are treated using *edge_order*

```
>>> x = np.array([0, 1, 2, 3, 4])
>>> f = x**2
>>> np.gradient(f, edge_order=1)
array([1.,  2.,  4.,  6.,  7.])
```

(continues on next page)

(continued from previous page)

```
>>> np.gradient(f, edge_order=2)
array([0., 2., 4., 6., 8.])
```

The *axis* keyword can be used to specify a subset of axes of which the gradient is calculated

```
>>> np.gradient(np.array([[1, 2, 6], [3, 4, 5]], dtype=float), axis=0)
array([[ 2.,  2., -1.],
       [ 2.,  2., -1.]])
```

`numpy.`**`cross`**(*a*, *b*, *axisa=-1*, *axisb=-1*, *axisc=-1*, *axis=None*)

Return the cross product of two (arrays of) vectors.

The cross product of *a* and *b* in $R^3$ is a vector perpendicular to both *a* and *b*. If *a* and *b* are arrays of vectors, the vectors are defined by the last axis of *a* and *b* by default, and these axes can have dimensions 2 or 3. Where the dimension of either *a* or *b* is 2, the third component of the input vector is assumed to be zero and the cross product calculated accordingly. In cases where both input vectors have dimension 2, the z-component of the cross product is returned.

**Parameters**

**a**

[array_like] Components of the first vector(s).

**b**

[array_like] Components of the second vector(s).

**axisa**

[int, optional] Axis of *a* that defines the vector(s). By default, the last axis.

**axisb**

[int, optional] Axis of *b* that defines the vector(s). By default, the last axis.

**axisc**

[int, optional] Axis of *c* containing the cross product vector(s). Ignored if both input vectors have dimension 2, as the return is scalar. By default, the last axis.

**axis**

[int, optional] If defined, the axis of *a*, *b* and *c* that defines the vector(s) and cross product(s). Overrides *axisa*, *axisb* and *axisc*.

**Returns**

**c**

[ndarray] Vector cross product(s).

**Raises**

**ValueError**

When the dimension of the vector(s) in *a* and/or *b* does not equal 2 or 3.

**See also:**

***inner***

Inner product

***outer***

Outer product.

***ix_***

Construct index arrays.

## Notes

New in version 1.9.0.

Supports full broadcasting of the inputs.

## Examples

Vector cross-product.

```
>>> x = [1, 2, 3]
>>> y = [4, 5, 6]
>>> np.cross(x, y)
array([-3,  6, -3])
```

One vector with dimension 2.

```
>>> x = [1, 2]
>>> y = [4, 5, 6]
>>> np.cross(x, y)
array([12, -6, -3])
```

Equivalently:

```
>>> x = [1, 2, 0]
>>> y = [4, 5, 6]
>>> np.cross(x, y)
array([12, -6, -3])
```

Both vectors with dimension 2.

```
>>> x = [1,2]
>>> y = [4,5]
>>> np.cross(x, y)
array(-3)
```

Multiple vector cross-products. Note that the direction of the cross product vector is defined by the *right-hand rule*.

```
>>> x = np.array([[1,2,3], [4,5,6]])
>>> y = np.array([[4,5,6], [1,2,3]])
>>> np.cross(x, y)
array([[-3,  6, -3],
       [ 3, -6,  3]])
```

The orientation of *c* can be changed using the *axisc* keyword.

```
>>> np.cross(x, y, axisc=0)
array([[-3,  3],
       [ 6, -6],
       [-3,  3]])
```

Change the vector definition of *x* and *y* using *axisa* and *axisb*.

```
>>> x = np.array([[1,2,3], [4,5,6], [7, 8, 9]])
>>> y = np.array([[7, 8, 9], [4,5,6], [1,2,3]])
>>> np.cross(x, y)
array([[ -6,  12,  -6],
       [  0,   0,   0],
       [  6, -12,   6]])
>>> np.cross(x, y, axisa=0, axisb=0)
array([[-24,  48, -24],
       [-30,  60, -30],
       [-36,  72, -36]])
```

`numpy.`**`trapz`**(*y*, *x=None*, *dx=1.0*, *axis=-1*)

Integrate along the given axis using the composite trapezoidal rule.

Integrate *y* (*x*) along given axis.

**Parameters**

**y**

[array_like] Input array to integrate.

**x**

[array_like, optional] The sample points corresponding to the *y* values. If *x* is None, the sample points are assumed to be evenly spaced *dx* apart. The default is None.

**dx**

[scalar, optional] The spacing between sample points when *x* is None. The default is 1.

**axis**

[int, optional] The axis along which to integrate.

**Returns**

**trapz**

[float] Definite integral as approximated by trapezoidal rule.

**See also:**

`sum`, `cumsum`

### Notes

Image [2] illustrates trapezoidal rule – y-axis locations of points will be taken from *y* array, by default x-axis distances between points will be 1.0, alternatively they can be provided with *x* array or with *dx* scalar. Return value will be equal to combined area under the red lines.

### References

[1], [2]

### Examples

```
>>> np.trapz([1,2,3])
4.0
>>> np.trapz([1,2,3], x=[4,6,8])
8.0
>>> np.trapz([1,2,3], dx=2)
8.0
>>> a = np.arange(6).reshape(2, 3)
>>> a
array([[0, 1, 2],
       [3, 4, 5]])
>>> np.trapz(a, axis=0)
array([1.5, 2.5, 3.5])
>>> np.trapz(a, axis=1)
array([2.,  8.])
```

## 4.19.5 Exponents and logarithms

| | |
|---|---|
| *exp*(x, /[, out, where, casting, order, …]) | Calculate the exponential of all elements in the input array. |
| *expm1*(x, /[, out, where, casting, order, …]) | Calculate `exp(x) - 1` for all elements in the array. |
| *exp2*(x, /[, out, where, casting, order, …]) | Calculate 2***p* for all *p* in the input array. |
| *log*(x, /[, out, where, casting, order, …]) | Natural logarithm, element-wise. |
| *log10*(x, /[, out, where, casting, order, …]) | Return the base 10 logarithm of the input array, element-wise. |
| *log2*(x, /[, out, where, casting, order, …]) | Base-2 logarithm of *x*. |
| *log1p*(x, /[, out, where, casting, order, …]) | Return the natural logarithm of one plus the input array, element-wise. |
| *logaddexp*(x1, x2, /[, out, where, casting, …]) | Logarithm of the sum of exponentiations of the inputs. |
| *logaddexp2*(x1, x2, /[, out, where, casting, …]) | Logarithm of the sum of exponentiations of the inputs in base-2. |

`numpy.`**`exp`**(*x, /, out=None, *, where=True, casting='same_kind', order='K', dtype=None, subok=True*[, *signature, extobj*]) **`= <ufunc 'exp'>`**

Calculate the exponential of all elements in the input array.

**Parameters**

**x**

[array_like] Input values.

**out**

[ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

**where**

[array_like, optional] This condition is broadcast over the input. At locations where the condition is True, the *out* array will be set to the ufunc result. Elsewhere, the *out* array will retain its original value. Note that if an uninitialized *out* array is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.

****kwargs**

For other keyword-only arguments, see the *ufunc docs*.

**Returns**

**out**

[ndarray or scalar] Output array, element-wise exponential of *x*. This is a scalar if *x* is a scalar.

**See also:**

**`expm1`**

Calculate `exp(x) - 1` for all elements in the array.

**`exp2`**

Calculate `2**x` for all elements in the array.

## Notes

The irrational number `e` is also known as Euler's number. It is approximately 2.718281, and is the base of the natural logarithm, `ln` (this means that, if $x = \ln y = \log_e y$, then $e^x = y$. For real input, `exp(x)` is always positive.

For complex arguments, `x = a + ib`, we can write $e^x = e^a e^{ib}$. The first term, $e^a$, is already known (it is the real argument, described above). The second term, $e^{ib}$, is $\cos b + i \sin b$, a function with magnitude 1 and a periodic phase.

## References

[1], [2]

## Examples

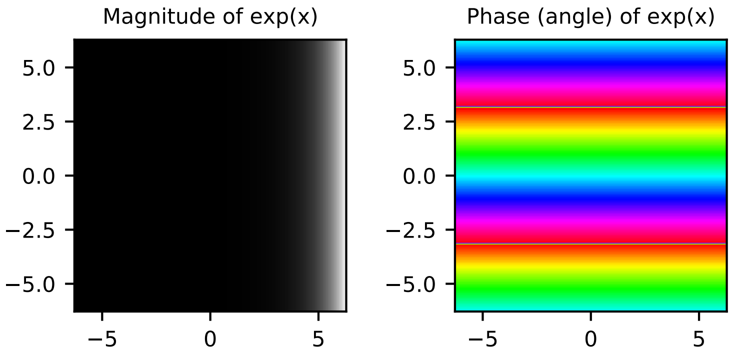
Plot the magnitude and phase of `exp(x)` in the complex plane:

```
>>> import matplotlib.pyplot as plt
```

```
>>> x = np.linspace(-2*np.pi, 2*np.pi, 100)
>>> xx = x + 1j * x[:, np.newaxis] # a + ib over complex plane
>>> out = np.exp(xx)
```

```
>>> plt.subplot(121)
>>> plt.imshow(np.abs(out),
...            extent=[-2*np.pi, 2*np.pi, -2*np.pi, 2*np.pi], cmap='gray')
>>> plt.title('Magnitude of exp(x)')
```

```
>>> plt.subplot(122)
>>> plt.imshow(np.angle(out),
...            extent=[-2*np.pi, 2*np.pi, -2*np.pi, 2*np.pi], cmap='hsv')
>>> plt.title('Phase (angle) of exp(x)')
>>> plt.show()
```



`numpy.`**`expm1`** (*x*, */*, *out=None*, *, *where=True*, *casting='same_kind'*, *order='K'*, *dtype=None*, *subok=True*[, *signature*, *extobj*]) **`= <ufunc 'expm1'>`**

Calculate `exp(x) - 1` for all elements in the array.

**Parameters**

**x**

[array_like] Input values.

**out**

[ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

**where**

[array_like, optional] This condition is broadcast over the input. At locations where the condition is True, the *out* array will be set to the ufunc result. Elsewhere, the *out* array will retain its original value. Note that if an uninitialized *out* array is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.

****kwargs**

For other keyword-only arguments, see the *ufunc docs*.

**Returns**

**out**

[ndarray or scalar] Element-wise exponential minus one: `out = exp(x) - 1`. This is a scalar if *x* is a scalar.

**See also:**

***log1p***

`log(1 + x)`, the inverse of expm1.

## Notes

This function provides greater precision than `exp(x) - 1` for small values of `x`.

## Examples

The true value of `exp(1e-10) - 1` is `1.00000000005e-10` to about 32 significant digits. This example shows the superiority of expm1 in this case.

```
>>> np.expm1(1e-10)
1.00000000005e-10
>>> np.exp(1e-10) - 1
1.000000082740371e-10
```

`numpy.`**`exp2`** (*x*, */*, *out=None*, ***, *where=True*, *casting='same_kind'*, *order='K'*, *dtype=None*, *subok=True*[, *signature*, *extobj* ]) **`= <ufunc 'exp2'>`**

Calculate *2**p* for all *p* in the input array.

**Parameters**

**x**

[array_like] Input values.

**out**

[ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

**where**

[array_like, optional] This condition is broadcast over the input. At locations where the condition is True, the *out* array will be set to the ufunc result. Elsewhere, the *out* array will retain

its original value. Note that if an uninitialized *out* array is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.

****kwargs**

For other keyword-only arguments, see the *ufunc docs*.

**Returns**

**out**

[ndarray or scalar] Element-wise 2 to the power *x*. This is a scalar if *x* is a scalar.

**See also:**

`power`

## Notes

New in version 1.3.0.

## Examples

```
>>> np.exp2([2, 3])
array([ 4.,  8.])
```

`numpy.log` (*x*, */*, *out=None*, ***, *where=True*, *casting='same_kind'*, *order='K'*, *dtype=None*, *subok=True*[, *signature*, *extobj*]) **= <ufunc 'log'>**

Natural logarithm, element-wise.

The natural logarithm `log` is the inverse of the exponential function, so that *log(exp(x))* = *x*. The natural logarithm is logarithm in base `e`.

**Parameters**

**x**

[array_like] Input value.

**out**

[ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

**where**

[array_like, optional] This condition is broadcast over the input. At locations where the condition is True, the *out* array will be set to the ufunc result. Elsewhere, the *out* array will retain its original value. Note that if an uninitialized *out* array is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.

****kwargs**

For other keyword-only arguments, see the *ufunc docs*.

**Returns**

**y**

[ndarray] The natural logarithm of *x*, element-wise. This is a scalar if *x* is a scalar.

**See also:**

*log10*, *log2*, *log1p*, `emath.log`

## Notes

Logarithm is a multivalued function: for each *x* there is an infinite number of *z* such that *exp(z) = x*. The convention is to return the *z* whose imaginary part lies in *[-pi, pi]*.

For real-valued input data types, *log* always returns real output. For each value that cannot be expressed as a real number or infinity, it yields `nan` and sets the *invalid* floating point error flag.

For complex-valued input, *log* is a complex analytical function that has a branch cut *[-inf, 0]* and is continuous from above on it. *log* handles the floating-point negative zero as an infinitesimal negative number, conforming to the C99 standard.

## References

[1], [2]

## Examples

```
>>> np.log([1, np.e, np.e**2, 0])
array([  0.,   1.,   2., -Inf])
```

`numpy.`**`log10`**(*x*, */*, *out=None*, ***, *where=True*, *casting='same_kind'*, *order='K'*, *dtype=None*, *subok=True*[, *signature*, *extobj*]) **`= <ufunc 'log10'>`**

Return the base 10 logarithm of the input array, element-wise.

**Parameters**

**x**

[array_like] Input values.

**out**

[ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

**where**

[array_like, optional] This condition is broadcast over the input. At locations where the condition is True, the *out* array will be set to the ufunc result. Elsewhere, the *out* array will retain its original value. Note that if an uninitialized *out* array is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.

****kwargs**

For other keyword-only arguments, see the *ufunc docs*.

**Returns**

**y**

[ndarray] The logarithm to the base 10 of *x*, element-wise. NaNs are returned where x is negative. This is a scalar if *x* is a scalar.

**See also:**

`emath.log10`

### Notes

Logarithm is a multivalued function: for each *x* there is an infinite number of *z* such that *10**z = x*. The convention is to return the *z* whose imaginary part lies in *[-pi, pi]*.

For real-valued input data types, *log10* always returns real output. For each value that cannot be expressed as a real number or infinity, it yields `nan` and sets the *invalid* floating point error flag.

For complex-valued input, *log10* is a complex analytical function that has a branch cut *[-inf, 0]* and is continuous from above on it. *log10* handles the floating-point negative zero as an infinitesimal negative number, conforming to the C99 standard.

### References

[1], [2]

### Examples

```
>>> np.log10([1e-15, -3.])
array([-15.,   nan])
```

`numpy.`**`log2`**(*x, /, out=None, *, where=True, casting='same_kind', order='K', dtype=None, subok=True*[*, signature, extobj*]) **`= <ufunc 'log2'>`**

Base-2 logarithm of *x*.

**Parameters**

**x**

[array_like] Input values.

**out**

[ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

**where**

[array_like, optional] This condition is broadcast over the input. At locations where the condition is True, the *out* array will be set to the ufunc result. Elsewhere, the *out* array will retain its original value. Note that if an uninitialized *out* array is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.

****kwargs**

For other keyword-only arguments, see the *ufunc docs*.

**Returns**

**y**

[ndarray] Base-2 logarithm of *x*. This is a scalar if *x* is a scalar.

**See also:**

*log*, *log10*, *log1p*, `emath.log2`

## Notes

New in version 1.3.0.

Logarithm is a multivalued function: for each *x* there is an infinite number of *z* such that *2**z = x*. The convention is to return the *z* whose imaginary part lies in *[-pi, pi]*.

For real-valued input data types, *log2* always returns real output. For each value that cannot be expressed as a real number or infinity, it yields `nan` and sets the *invalid* floating point error flag.

For complex-valued input, *log2* is a complex analytical function that has a branch cut *[-inf, 0]* and is continuous from above on it. *log2* handles the floating-point negative zero as an infinitesimal negative number, conforming to the C99 standard.

## Examples

```
>>> x = np.array([0, 1, 2, 2**4])
>>> np.log2(x)
array([-Inf,   0.,   1.,   4.])
```

```
>>> xi = np.array([0+1.j, 1, 2+0.j, 4.j])
>>> np.log2(xi)
array([ 0.+2.26618007j,  0.+0.j        ,  1.+0.j        ,  2.+2.26618007j])
```

`numpy.`**`log1p`**(*x*, */*, *out=None*, ***, *where=True*, *casting='same_kind'*, *order='K'*, *dtype=None*, *subok=True*[, *signature*, *extobj*]) **`= <ufunc 'log1p'>`**

Return the natural logarithm of one plus the input array, element-wise.

Calculates `log(1 + x)`.

**Parameters**

**x**

[array_like] Input values.

**out**

[ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

**where**

[array_like, optional] This condition is broadcast over the input. At locations where the condition is True, the *out* array will be set to the ufunc result. Elsewhere, the *out* array will retain its original value. Note that if an uninitialized *out* array is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.

****kwargs**

For other keyword-only arguments, see the *ufunc docs*.

**Returns**

**y**

[ndarray] Natural logarithm of *1 + x*, element-wise. This is a scalar if *x* is a scalar.

**See also:**

***expm1***

`exp(x) - 1`, the inverse of *log1p*.

## Notes

For real-valued input, *log1p* is accurate also for *x* so small that *1 + x == 1* in floating-point accuracy.

Logarithm is a multivalued function: for each *x* there is an infinite number of *z* such that *exp(z) = 1 + x*. The convention is to return the *z* whose imaginary part lies in *[-pi, pi]*.

For real-valued input data types, *log1p* always returns real output. For each value that cannot be expressed as a real number or infinity, it yields `nan` and sets the *invalid* floating point error flag.

For complex-valued input, *log1p* is a complex analytical function that has a branch cut *[-inf, -1]* and is continuous from above on it. *log1p* handles the floating-point negative zero as an infinitesimal negative number, conforming to the C99 standard.

## References

[1], [2]

## Examples

```
>>> np.log1p(1e-99)
1e-99
>>> np.log(1 + 1e-99)
0.0
```

`numpy.`**`logaddexp`**(*x1, x2, /, out=None, *, where=True, casting='same_kind', order='K', dtype=None, subok=True*[*, signature, extobj*]) **= <ufunc 'logaddexp'>**

Logarithm of the sum of exponentiations of the inputs.

Calculates `log(exp(x1) + exp(x2))`. This function is useful in statistics where the calculated probabilities of events may be so small as to exceed the range of normal floating point numbers. In such cases the logarithm of the calculated probability is stored. This function allows adding probabilities stored in such a fashion.

**Parameters**

**x1, x2**

[array_like] Input values. If `x1.shape != x2.shape`, they must be broadcastable to a common shape (which becomes the shape of the output).

**out**

[ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

**where**

[array_like, optional] This condition is broadcast over the input. At locations where the condition is True, the *out* array will be set to the ufunc result. Elsewhere, the *out* array will retain its original value. Note that if an uninitialized *out* array is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.

****kwargs**

For other keyword-only arguments, see the *ufunc docs*.

**Returns**

**result**

[ndarray] Logarithm of `exp(x1) + exp(x2)`. This is a scalar if both *x1* and *x2* are scalars.

**See also:**

`logaddexp2`

Logarithm of the sum of exponentiations of inputs in base 2.

### Notes

New in version 1.3.0.

### Examples

```
>>> prob1 = np.log(1e-50)
>>> prob2 = np.log(2.5e-50)
>>> prob12 = np.logaddexp(prob1, prob2)
>>> prob12
-113.87649168120691
>>> np.exp(prob12)
3.5000000000000057e-50
```

`numpy.`**`logaddexp2`**(*x1*, *x2*, */*, *out=None*, ***, *where=True*, *casting='same_kind'*, *order='K'*, *dtype=None*, *subok=True*[, *signature*, *extobj* ]) **`= <ufunc 'logaddexp2'>`**

Logarithm of the sum of exponentiations of the inputs in base-2.

Calculates `log2(2**x1 + 2**x2)`. This function is useful in machine learning when the calculated probabilities of events may be so small as to exceed the range of normal floating point numbers. In such cases the base-2 logarithm of the calculated probability can be used instead. This function allows adding probabilities stored in such a fashion.

**Parameters**

**x1, x2**

[array_like] Input values. If `x1.shape != x2.shape`, they must be broadcastable to a common shape (which becomes the shape of the output).

**out**

[ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

**where**

[array_like, optional] This condition is broadcast over the input. At locations where the condition is True, the *out* array will be set to the ufunc result. Elsewhere, the *out* array will retain its original value. Note that if an uninitialized *out* array is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.

****kwargs**

For other keyword-only arguments, see the *ufunc docs*.

**Returns**

**result**

[ndarray] Base-2 logarithm of `2**x1 + 2**x2`. This is a scalar if both *x1* and *x2* are scalars.

**See also:**

**`logaddexp`**

Logarithm of the sum of exponentiations of the inputs.

## Notes

New in version 1.3.0.

## Examples

```
>>> prob1 = np.log2(1e-50)
>>> prob2 = np.log2(2.5e-50)
>>> prob12 = np.logaddexp2(prob1, prob2)
>>> prob1, prob2, prob12
(-166.09640474436813, -164.77447664948076, -164.28904982231052)
>>> 2**prob12
3.4999999999999914e-50
```

## 4.19.6 Other special functions

| | |
|---|---|
| *i0*(x) | Modified Bessel function of the first kind, order 0. |
| *sinc*(x) | Return the sinc function. |

`numpy.`**`i0`**(*x*)

Modified Bessel function of the first kind, order 0.

Usually denoted $I_0$. This function does broadcast, but will *not* "up-cast" int dtype arguments unless accompanied by at least one float or complex dtype argument (see Raises below).

**Parameters**

**x**

[array_like, dtype float or complex] Argument of the Bessel function.

**Returns**

**out**

[ndarray, shape = x.shape, dtype = x.dtype] The modified Bessel function evaluated at each of the elements of *x*.

**Raises**

**TypeError: array cannot be safely cast to required type**

If argument consists exclusively of int dtypes.

**See also:**

`scipy.special.i0`, `scipy.special.iv`, `scipy.special.ive`

### Notes

The scipy implementation is recommended over this function: it is a proper ufunc written in C, and more than an order of magnitude faster.

We use the algorithm published by Clenshaw [1] and referenced by Abramowitz and Stegun [2], for which the function domain is partitioned into the two intervals [0,8] and (8,inf), and Chebyshev polynomial expansions are employed in each interval. Relative error on the domain [0,30] using IEEE arithmetic is documented [3] as having a peak of 5.8e-16 with an rms of 1.4e-16 (n = 30000).

### References

[1], [2], [3]

### Examples

```
>>> np.i0(0.)
array(1.0)  # may vary
>>> np.i0([0., 1. + 2j])
array([ 1.00000000+0.j        ,  0.18785373+0.64616944j])  # may vary
```

numpy.**sinc**(*x*)

Return the sinc function.

The sinc function is $\sin(\pi x)/(\pi x)$.

**Parameters**

**x**

[ndarray] Array (possibly multi-dimensional) of values for which to to calculate `sinc(x)`.

**Returns**

**out**

[ndarray] `sinc(x)`, which has the same shape as the input.

### Notes

`sinc(0)` is the limit value 1.

The name sinc is short for "sine cardinal" or "sinus cardinalis".

The sinc function is used in various signal processing applications, including in anti-aliasing, in the construction of a Lanczos resampling filter, and in interpolation.

For bandlimited interpolation of discrete-time signals, the ideal interpolation kernel is proportional to the sinc function.

### References

[1], [2]

### Examples

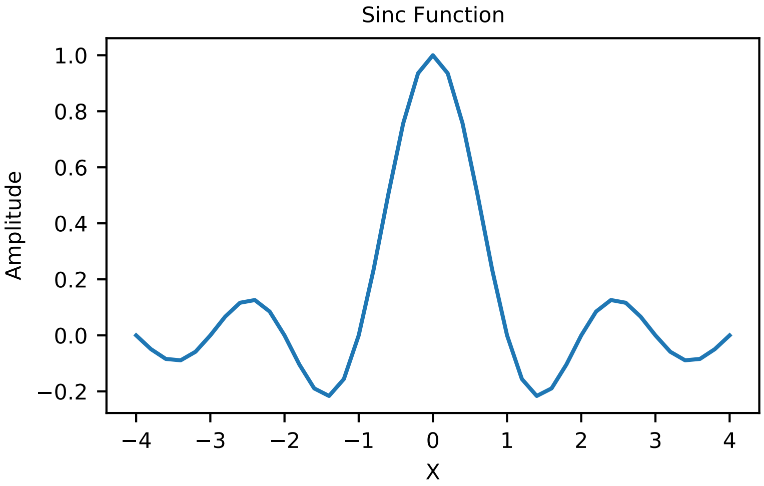
```
>>> import matplotlib.pyplot as plt
>>> x = np.linspace(-4, 4, 41)
>>> np.sinc(x)
 array([-3.89804309e-17,  -4.92362781e-02,  -8.40918587e-02, # may vary
        -8.90384387e-02,  -5.84680802e-02,   3.89804309e-17,
        6.68206631e-02,   1.16434881e-01,   1.26137788e-01,
        8.50444803e-02,  -3.89804309e-17,  -1.03943254e-01,
        -1.89206682e-01,  -2.16236208e-01,  -1.55914881e-01,
        3.89804309e-17,   2.33872321e-01,   5.04551152e-01,
        7.56826729e-01,   9.35489284e-01,   1.00000000e+00,
        9.35489284e-01,   7.56826729e-01,   5.04551152e-01,
        2.33872321e-01,   3.89804309e-17,  -1.55914881e-01,
       -2.16236208e-01,  -1.89206682e-01,  -1.03943254e-01,
```

(continues on next page)

(continued from previous page)

```
       -3.89804309e-17,   8.50444803e-02,   1.26137788e-01,
        1.16434881e-01,   6.68206631e-02,   3.89804309e-17,
        -5.84680802e-02,  -8.90384387e-02,  -8.40918587e-02,
        -4.92362781e-02,  -3.89804309e-17])
```

```
>>> plt.plot(x, np.sinc(x))
[<matplotlib.lines.Line2D object at 0x...>]
>>> plt.title("Sinc Function")
Text(0.5, 1.0, 'Sinc Function')
>>> plt.ylabel("Amplitude")
Text(0, 0.5, 'Amplitude')
>>> plt.xlabel("X")
Text(0.5, 0, 'X')
>>> plt.show()
```



## 4.19.7 Floating point routines

| | |
|---|---|
| *signbit*(x, /[, out, where, casting, order, …]) | Returns element-wise True where signbit is set (less than zero). |
| *copysign*(x1, x2, /[, out, where, casting, …]) | Change the sign of x1 to that of x2, element-wise. |
| *frexp*(x[, out1, out2], / [[, out, where, …]) | Decompose the elements of x into mantissa and twos exponent. |
| *ldexp*(x1, x2, /[, out, where, casting, …]) | Returns x1 * 2**x2, element-wise. |
| *nextafter*(x1, x2, /[, out, where, casting, …]) | Return the next floating-point value after x1 towards x2, element-wise. |
| *spacing*(x, /[, out, where, casting, order, …]) | Return the distance between x and the nearest adjacent number. |

`numpy.`**`signbit`** (*x*, */*, *out=None*, ***, *where=True*, *casting='same_kind'*, *order='K'*, *dtype=None*, *subok=True*[, *signature*, *extobj*]) **`= <ufunc 'signbit'>`**

Returns element-wise True where signbit is set (less than zero).

**Parameters**

**x**

[array_like] The input value(s).

**out**

[ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

**where**

[array_like, optional] This condition is broadcast over the input. At locations where the condition is True, the *out* array will be set to the ufunc result. Elsewhere, the *out* array will retain its original value. Note that if an uninitialized *out* array is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.

****kwargs**

For other keyword-only arguments, see the *ufunc docs*.

**Returns**

**result**

[ndarray of bool] Output array, or reference to *out* if that was supplied. This is a scalar if *x* is a scalar.

### Examples

```
>>> np.signbit(-1.2)
True
>>> np.signbit(np.array([1, -2.3, 2.1]))
array([False,  True, False])
```

`numpy.`**`copysign`**(*x1*, *x2*, */*, *out=None*, ***, *where=True*, *casting='same_kind'*, *order='K'*, *dtype=None*, *subok=True*[, *signature*, *extobj* ]) **`= <ufunc 'copysign'>`**

Change the sign of x1 to that of x2, element-wise.

If *x2* is a scalar, its sign will be copied to all elements of *x1*.

**Parameters**

**x1**

[array_like] Values to change the sign of.

**x2**

[array_like] The sign of *x2* is copied to *x1*. If `x1.shape != x2.shape`, they must be broadcastable to a common shape (which becomes the shape of the output).

**out**

[ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

**where**

[array_like, optional] This condition is broadcast over the input. At locations where the condition is True, the *out* array will be set to the ufunc result. Elsewhere, the *out* array will retain its original value. Note that if an uninitialized *out* array is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.

****kwargs**

For other keyword-only arguments, see the *ufunc docs*.

**Returns**

**out**

[ndarray or scalar] The values of *x1* with the sign of *x2*. This is a scalar if both *x1* and *x2* are scalars.

## Examples

```
>>> np.copysign(1.3, -1)
-1.3
>>> 1/np.copysign(0, 1)
inf
>>> 1/np.copysign(0, -1)
-inf
```

```
>>> np.copysign([-1, 0, 1], -1.1)
array([-1., -0., -1.])
>>> np.copysign([-1, 0, 1], np.arange(3)-1)
array([-1.,  0.,  1.])
```

`numpy.`**`frexp`**(*x*[, *out1*, *out2* ], /[, *out=(None, None)* ], *, *where=True*, *casting='same_kind'*, *order='K'*, *dtype=None*, *subok=True*[, *signature*, *extobj* ]) **`= <ufunc 'frexp'>`**

Decompose the elements of x into mantissa and twos exponent.

Returns (*mantissa*, *exponent*), where *x = mantissa * 2**exponent`*. The mantissa is lies in the open interval(-1, 1), while the twos exponent is a signed integer.

**Parameters**

**x**

[array_like] Array of numbers to be decomposed.

**out1**

[ndarray, optional] Output array for the mantissa. Must have the same shape as *x*.

**out2**

[ndarray, optional] Output array for the exponent. Must have the same shape as *x*.

**out**

[ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

**where**

[array_like, optional] This condition is broadcast over the input. At locations where the condition is True, the *out* array will be set to the ufunc result. Elsewhere, the *out* array will retain its original value. Note that if an uninitialized *out* array is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.

****kwargs**

For other keyword-only arguments, see the *ufunc docs*.

**Returns**

**mantissa**

[ndarray] Floating values between -1 and 1. This is a scalar if *x* is a scalar.

**exponent**

[ndarray] Integer exponents of 2. This is a scalar if *x* is a scalar.

**See also:**

**`ldexp`**

Compute `y = x1 * 2**x2`, the inverse of `frexp`.

## Notes

Complex dtypes are not supported, they will raise a TypeError.

## Examples

```
>>> x = np.arange(9)
>>> y1, y2 = np.frexp(x)
>>> y1
array([ 0.   ,  0.5  ,  0.5  ,  0.75 ,  0.5  ,  0.625,  0.75 ,  0.875,
        0.5  ])
>>> y2
array([0, 1, 2, 2, 3, 3, 3, 3, 4])
>>> y1 * 2**y2
array([ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.])
```

`numpy.`**`ldexp`**(*x1*, *x2*, */*, *out=None*, ***, *where=True*, *casting='same_kind'*, *order='K'*, *dtype=None*, *subok=True*[, *signature*, *extobj*]) **`= <ufunc 'ldexp'>`**

Returns x1 * 2**x2, element-wise.

The mantissas *x1* and twos exponents *x2* are used to construct floating point numbers `x1 * 2**x2`.

**Parameters**

**x1**

[array_like] Array of multipliers.

**x2**

[array_like, int] Array of twos exponents. If `x1.shape != x2.shape`, they must be broadcastable to a common shape (which becomes the shape of the output).

**out**

[ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

**where**

[array_like, optional] This condition is broadcast over the input. At locations where the condition is True, the *out* array will be set to the ufunc result. Elsewhere, the *out* array will retain its original value. Note that if an uninitialized *out* array is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.

****kwargs**

For other keyword-only arguments, see the *ufunc docs*.

**Returns**

**y**

[ndarray or scalar] The result of `x1 * 2**x2`. This is a scalar if both *x1* and *x2* are scalars.

**See also:**

***`frexp`***

Return (y1, y2) from `x = y1 * 2**y2`, inverse to *`ldexp`*.

## Notes

Complex dtypes are not supported, they will raise a TypeError.

*`ldexp`* is useful as the inverse of *`frexp`*, if used by itself it is more clear to simply use the expression `x1 * 2**x2`.

## Examples

```
>>> np.ldexp(5, np.arange(4))
array([ 5., 10., 20., 40.], dtype=float16)
```

```
>>> x = np.arange(6)
>>> np.ldexp(*np.frexp(x))
array([ 0.,  1.,  2.,  3.,  4.,  5.])
```

`numpy.`**`nextafter`**(*x1*, *x2*, */*, *out=None*, ***, *where=True*, *casting='same_kind'*, *order='K'*, *dtype=None*, *subok=True*[, *signature*, *extobj* ]) **`= <ufunc 'nextafter'>`**

Return the next floating-point value after x1 towards x2, element-wise.

**Parameters**

**x1**

[array_like] Values to find the next representable value of.

**x2**

[array_like] The direction where to look for the next representable value of *x1*. If `x1.shape != x2.shape`, they must be broadcastable to a common shape (which becomes the shape of the output).

**out**

[ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

**where**

[array_like, optional] This condition is broadcast over the input. At locations where the condition is True, the *out* array will be set to the ufunc result. Elsewhere, the *out* array will retain its original value. Note that if an uninitialized *out* array is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.

****kwargs**

For other keyword-only arguments, see the *ufunc docs*.

**Returns**

**out**

[ndarray or scalar] The next representable values of *x1* in the direction of *x2*. This is a scalar if both *x1* and *x2* are scalars.

## Examples

```
>>> eps = np.finfo(np.float64).eps
>>> np.nextafter(1, 2) == eps + 1
True
>>> np.nextafter([1, 2], [2, 1]) == [eps + 1, 2 - eps]
array([ True,  True])
```

`numpy.`**`spacing`**(*x*, */*, *out=None*, ***, *where=True*, *casting='same_kind'*, *order='K'*, *dtype=None*, *subok=True*[, *signature*, *extobj* ]) **`= <ufunc 'spacing'>`**

Return the distance between x and the nearest adjacent number.

**Parameters**

**x**

[array_like] Values to find the spacing of.

**out**

[ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

**where**

[array_like, optional] This condition is broadcast over the input. At locations where the condition is True, the *out* array will be set to the ufunc result. Elsewhere, the *out* array will retain

its original value. Note that if an uninitialized *out* array is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.

****kwargs**

For other keyword-only arguments, see the *ufunc docs*.

**Returns**

**out**

[ndarray or scalar] The spacing of values of *x*. This is a scalar if *x* is a scalar.

### Notes

It can be considered as a generalization of EPS: `spacing(np.float64(1)) == np.finfo(np.float64).eps`, and there should not be any representable number between `x + spacing(x)` and x for any finite x.

Spacing of +- inf and NaN is NaN.

### Examples

```
>>> np.spacing(1) == np.finfo(np.float64).eps
True
```

## 4.19.8 Rational routines

| | |
|---|---|
| *`lcm`*(x1, x2, /[, out, where, casting, order, …]) | Returns the lowest common multiple of `|x1|` and `|x2|` |
| *`gcd`*(x1, x2, /[, out, where, casting, order, …]) | Returns the greatest common divisor of `|x1|` and `|x2|` |

`numpy.`**`lcm`**(*x1*, *x2*, */*, *out=None*, ***, *where=True*, *casting='same_kind'*, *order='K'*, *dtype=None*, *subok=True*[, *signature*, *extobj*]) **`= <ufunc 'lcm'>`**

Returns the lowest common multiple of `|x1|` and `|x2|`

**Parameters**

**x1, x2**

[array_like, int] Arrays of values. If `x1.shape != x2.shape`, they must be broadcastable to a common shape (which becomes the shape of the output).

**Returns**

**y**

[ndarray or scalar] The lowest common multiple of the absolute value of the inputs This is a scalar if both *x1* and *x2* are scalars.

**See also:**

*`gcd`*

The greatest common divisor

### Examples

```
>>> np.lcm(12, 20)
60
>>> np.lcm.reduce([3, 12, 20])
60
>>> np.lcm.reduce([40, 12, 20])
120
>>> np.lcm(np.arange(6), 20)
array([ 0, 20, 20, 60, 20, 20])
```

`numpy.`**`gcd`**(*x1*, *x2*, /, *out=None*, *, *where=True*, *casting='same_kind'*, *order='K'*, *dtype=None*, *subok=True*[, *signature*, *extobj*]) **`= <ufunc 'gcd'>`**

Returns the greatest common divisor of `|x1|` and `|x2|`

**Parameters**

**x1, x2**

[array_like, int] Arrays of values. If `x1.shape != x2.shape`, they must be broadcastable to a common shape (which becomes the shape of the output).

**Returns**

**y**

[ndarray or scalar] The greatest common divisor of the absolute value of the inputs This is a scalar if both *x1* and *x2* are scalars.

**See also:**

*`lcm`*

The lowest common multiple

### Examples

```
>>> np.gcd(12, 20)
4
>>> np.gcd.reduce([15, 25, 35])
5
>>> np.gcd(np.arange(6), 20)
array([20,  1,  2,  1,  4,  5])
```

## 4.19.9 Arithmetic operations

| | |
|---|---|
| *`add`*(x1, x2, /[, out, where, casting, order, …]) | Add arguments element-wise. |
| *`reciprocal`*(x, /[, out, where, casting, …]) | Return the reciprocal of the argument, element-wise. |
| *`positive`*(x, /[, out, where, casting, order, …]) | Numerical positive, element-wise. |
| *`negative`*(x, /[, out, where, casting, order, …]) | Numerical negative, element-wise. |
| *`multiply`*(x1, x2, /[, out, where, casting, …]) | Multiply arguments element-wise. |
| *`divide`*(x1, x2, /[, out, where, casting, …]) | Returns a true division of the inputs, element-wise. |

Continued on next page

Table 85 – continued from previous page

| | |
|---|---|
| `power`(x1, x2, /[, out, where, casting, …]) | First array elements raised to powers from second array, element-wise. |
| `subtract`(x1, x2, /[, out, where, casting, …]) | Subtract arguments, element-wise. |
| `true_divide`(x1, x2, /[, out, where, …]) | Returns a true division of the inputs, element-wise. |
| `floor_divide`(x1, x2, /[, out, where, …]) | Return the largest integer smaller or equal to the division of the inputs. |
| `float_power`(x1, x2, /[, out, where, …]) | First array elements raised to powers from second array, element-wise. |
| `fmod`(x1, x2, /[, out, where, casting, …]) | Return the element-wise remainder of division. |
| `mod`(x1, x2, /[, out, where, casting, order, …]) | Return element-wise remainder of division. |
| `modf`(x[, out1, out2], / [[, out, where, …]) | Return the fractional and integral parts of an array, element-wise. |
| `remainder`(x1, x2, /[, out, where, casting, …]) | Return element-wise remainder of division. |
| `divmod`(x1, x2[, out1, out2], / [[, out, …]) | Return element-wise quotient and remainder simultaneously. |

`numpy.`**`add`**(*x1*, *x2*, */*, *out=None*, ***, *where=True*, *casting='same_kind'*, *order='K'*, *dtype=None*, *subok=True*[, *signature*, *extobj*]) **`= <ufunc 'add'>`**

Add arguments element-wise.

**Parameters**

**x1, x2**

[array_like] The arrays to be added. If `x1.shape != x2.shape`, they must be broadcastable to a common shape (which becomes the shape of the output).

**out**

[ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

**where**

[array_like, optional] This condition is broadcast over the input. At locations where the condition is True, the *out* array will be set to the ufunc result. Elsewhere, the *out* array will retain its original value. Note that if an uninitialized *out* array is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.

****kwargs**

For other keyword-only arguments, see the *ufunc docs*.

**Returns**

**add**

[ndarray or scalar] The sum of *x1* and *x2*, element-wise. This is a scalar if both *x1* and *x2* are scalars.

### Notes

Equivalent to *x1* + *x2* in terms of array broadcasting.

### Examples

```
>>> np.add(1.0, 4.0)
5.0
>>> x1 = np.arange(9.0).reshape((3, 3))
>>> x2 = np.arange(3.0)
>>> np.add(x1, x2)
array([[  0.,   2.,   4.],
       [  3.,   5.,   7.],
       [  6.,   8.,  10.]])
```

`numpy.`**`reciprocal`**(*x*, */*, *out=None*, ***, *where=True*, *casting='same_kind'*, *order='K'*, *dtype=None*, *subok=True*[, *signature*, *extobj*]) **`= <ufunc 'reciprocal'>`**

Return the reciprocal of the argument, element-wise.

Calculates `1/x`.

**Parameters**

**x**

[array_like] Input array.

**out**

[ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

**where**

[array_like, optional] This condition is broadcast over the input. At locations where the condition is True, the *out* array will be set to the ufunc result. Elsewhere, the *out* array will retain its original value. Note that if an uninitialized *out* array is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.

****kwargs**

For other keyword-only arguments, see the *ufunc docs*.

**Returns**

**y**

[ndarray] Return array. This is a scalar if *x* is a scalar.

### Notes

---

**Note:** This function is not designed to work with integers.

---

For integer arguments with absolute value larger than 1 the result is always zero because of the way Python handles integer division. For integer zero the result is an overflow.

### Examples

```
>>> np.reciprocal(2.)
0.5
>>> np.reciprocal([1, 2., 3.33])
array([ 1.       ,  0.5      ,  0.3003003])
```

`numpy.`**`positive`**(*x*, */*, *out=None*, ***, *where=True*, *casting='same_kind'*, *order='K'*, *dtype=None*, *subok=True*[, *signature*, *extobj* ]) **= <ufunc 'positive'>**

Numerical positive, element-wise.

New in version 1.13.0.

**Parameters**

**x**

[array_like or scalar] Input array.

**Returns**

**y**

[ndarray or scalar] Returned array or scalar: $y = +x$. This is a scalar if $x$ is a scalar.

### Notes

Equivalent to *x.copy()*, but only defined for types that support arithmetic.

`numpy.`**`negative`**(*x*, */*, *out=None*, ***, *where=True*, *casting='same_kind'*, *order='K'*, *dtype=None*, *subok=True*[, *signature*, *extobj* ]) **= <ufunc 'negative'>**

Numerical negative, element-wise.

**Parameters**

**x**

[array_like or scalar] Input array.

**out**

[ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

**where**

[array_like, optional] This condition is broadcast over the input. At locations where the condition is True, the *out* array will be set to the ufunc result. Elsewhere, the *out* array will retain

its original value. Note that if an uninitialized *out* array is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.

****kwargs**

For other keyword-only arguments, see the *ufunc docs*.

**Returns**

**y**

[ndarray or scalar] Returned array or scalar: *y = -x*. This is a scalar if *x* is a scalar.

## Examples

```
>>> np.negative([1.,-1.])
array([-1.,  1.])
```

numpy.**multiply**(*x1*, *x2*, */*, *out=None*, *, *where=True*, *casting='same_kind'*, *order='K'*, *dtype=None*, *subok=True*[, *signature*, *extobj*]) **= <ufunc 'multiply'>**

Multiply arguments element-wise.

**Parameters**

**x1, x2**

[array_like] Input arrays to be multiplied. If `x1.shape != x2.shape`, they must be broadcastable to a common shape (which becomes the shape of the output).

**out**

[ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

**where**

[array_like, optional] This condition is broadcast over the input. At locations where the condition is True, the *out* array will be set to the ufunc result. Elsewhere, the *out* array will retain its original value. Note that if an uninitialized *out* array is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.

****kwargs**

For other keyword-only arguments, see the *ufunc docs*.

**Returns**

**y**

[ndarray] The product of *x1* and *x2*, element-wise. This is a scalar if both *x1* and *x2* are scalars.

## Notes

Equivalent to *x1* * *x2* in terms of array broadcasting.

## Examples

```
>>> np.multiply(2.0, 4.0)
8.0
```

```
>>> x1 = np.arange(9.0).reshape((3, 3))
>>> x2 = np.arange(3.0)
>>> np.multiply(x1, x2)
array([[  0.,   1.,   4.],
       [  0.,   4.,  10.],
       [  0.,   7.,  16.]])
```

`numpy.`**`divide`**(*x1*, *x2*, */*, *out=None*, ***, *where=True*, *casting='same_kind'*, *order='K'*, *dtype=None*, *subok=True*[, *signature*, *extobj* ]) **`= <ufunc 'true_divide'>`**

Returns a true division of the inputs, element-wise.

Instead of the Python traditional 'floor division', this returns a true division. True division adjusts the output type to present the best answer, regardless of input types.

**Parameters**

**x1**

[array_like] Dividend array.

**x2**

[array_like] Divisor array. If `x1.shape != x2.shape`, they must be broadcastable to a common shape (which becomes the shape of the output).

**out**

[ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

**where**

[array_like, optional] This condition is broadcast over the input. At locations where the condition is True, the *out* array will be set to the ufunc result. Elsewhere, the *out* array will retain its original value. Note that if an uninitialized *out* array is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.

****kwargs**

For other keyword-only arguments, see the *ufunc docs*.

**Returns**

**out**

[ndarray or scalar] This is a scalar if both *x1* and *x2* are scalars.

## Notes

In Python, `//` is the floor division operator and `/` the true division operator. The `true_divide(x1, x2)` function is equivalent to true division in Python.

## Examples

```
>>> x = np.arange(5)
>>> np.true_divide(x, 4)
array([ 0.  ,  0.25,  0.5 ,  0.75,  1.  ])
```

```
>>> x/4
array([ 0.  ,  0.25,  0.5 ,  0.75,  1.  ])
```

```
>>> x//4
array([0, 0, 0, 0, 1])
```

`numpy.`**`power`**(*x1*, *x2*, */*, *out=None*, ***, *where=True*, *casting='same_kind'*, *order='K'*, *dtype=None*, *subok=True*[, *signature*, *extobj*]) **`= <ufunc 'power'>`**

First array elements raised to powers from second array, element-wise.

Raise each base in *x1* to the positionally-corresponding power in *x2*. *x1* and *x2* must be broadcastable to the same shape. Note that an integer type raised to a negative integer power will raise a ValueError.

**Parameters**

**x1**

[array_like] The bases.

**x2**

[array_like] The exponents. If `x1.shape != x2.shape`, they must be broadcastable to a common shape (which becomes the shape of the output).

**out**

[ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

**where**

[array_like, optional] This condition is broadcast over the input. At locations where the condition is True, the *out* array will be set to the ufunc result. Elsewhere, the *out* array will retain its original value. Note that if an uninitialized *out* array is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.

****kwargs**

For other keyword-only arguments, see the *ufunc docs*.

**Returns**

**y**

[ndarray] The bases in *x1* raised to the exponents in *x2*. This is a scalar if both *x1* and *x2* are scalars.

**See also:**

*float_power*

power function that promotes integers to float

### Examples

Cube each element in a list.

```
>>> x1 = range(6)
>>> x1
[0, 1, 2, 3, 4, 5]
>>> np.power(x1, 3)
array([  0,   1,   8,  27,  64, 125])
```

Raise the bases to different exponents.

```
>>> x2 = [1.0, 2.0, 3.0, 3.0, 2.0, 1.0]
>>> np.power(x1, x2)
array([  0.,   1.,   8.,  27.,  16.,   5.])
```

The effect of broadcasting.

```
>>> x2 = np.array([[1, 2, 3, 3, 2, 1], [1, 2, 3, 3, 2, 1]])
>>> x2
array([[1, 2, 3, 3, 2, 1],
       [1, 2, 3, 3, 2, 1]])
>>> np.power(x1, x2)
array([[ 0,  1,  8, 27, 16,  5],
       [ 0,  1,  8, 27, 16,  5]])
```

`numpy.`**`subtract`**(*x1*, *x2*, */*, *out=None*, ***, *where=True*, *casting='same_kind'*, *order='K'*, *dtype=None*, *subok=True*[, *signature*, *extobj*]) **`= <ufunc 'subtract'>`**

Subtract arguments, element-wise.

**Parameters**

**x1, x2**

[array_like] The arrays to be subtracted from each other. If `x1.shape != x2.shape`, they must be broadcastable to a common shape (which becomes the shape of the output).

**out**

[ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

**where**

[array_like, optional] This condition is broadcast over the input. At locations where the condition is True, the *out* array will be set to the ufunc result. Elsewhere, the *out* array will retain its original value. Note that if an uninitialized *out* array is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.

****kwargs**

For other keyword-only arguments, see the *ufunc docs*.

**Returns**

**y**

[ndarray] The difference of *x1* and *x2*, element-wise. This is a scalar if both *x1* and *x2* are scalars.

## Notes

Equivalent to `x1 - x2` in terms of array broadcasting.

## Examples

```
>>> np.subtract(1.0, 4.0)
-3.0
```

```
>>> x1 = np.arange(9.0).reshape((3, 3))
>>> x2 = np.arange(3.0)
>>> np.subtract(x1, x2)
array([[ 0.,   0.,   0.],
       [ 3.,   3.,   3.],
       [ 6.,   6.,   6.]])
```

`numpy.`**`true_divide`**(*x1*, *x2*, */*, *out=None*, ***, *where=True*, *casting='same_kind'*, *order='K'*, *dtype=None*, *subok=True*[, *signature*, *extobj* ]) **`= <ufunc 'true_divide'>`**

Returns a true division of the inputs, element-wise.

Instead of the Python traditional 'floor division', this returns a true division. True division adjusts the output type to present the best answer, regardless of input types.

**Parameters**

**x1**

[array_like] Dividend array.

**x2**

[array_like] Divisor array. If `x1.shape != x2.shape`, they must be broadcastable to a common shape (which becomes the shape of the output).

**out**

[ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

**where**

[array_like, optional] This condition is broadcast over the input. At locations where the condition is True, the *out* array will be set to the ufunc result. Elsewhere, the *out* array will retain its original value. Note that if an uninitialized *out* array is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.

****kwargs**

For other keyword-only arguments, see the *ufunc docs*.

**Returns**

**out**

[ndarray or scalar] This is a scalar if both *x1* and *x2* are scalars.

## Notes

In Python, `//` is the floor division operator and `/` the true division operator. The `true_divide(x1, x2)` function is equivalent to true division in Python.

## Examples

```
>>> x = np.arange(5)
>>> np.true_divide(x, 4)
array([ 0.  ,  0.25,  0.5 ,  0.75,  1.  ])
```

```
>>> x/4
array([ 0.  ,  0.25,  0.5 ,  0.75,  1.  ])
```

```
>>> x//4
array([0, 0, 0, 0, 1])
```

`numpy.`**`floor_divide`**(*x1*, *x2*, */*, *out=None*, ***, *where=True*, *casting='same_kind'*, *order='K'*, *dtype=None*, *subok=True*[, *signature*, *extobj* ]) **`= <ufunc 'floor_divide'>`**

Return the largest integer smaller or equal to the division of the inputs. It is equivalent to the Python `//` operator and pairs with the Python `%` (*remainder*), function so that `a = a % b + b * (a // b)` up to roundoff.

**Parameters**

**x1**

[array_like] Numerator.

**x2**

[array_like] Denominator. If `x1.shape != x2.shape`, they must be broadcastable to a common shape (which becomes the shape of the output).

**out**

[ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

**where**

[array_like, optional] This condition is broadcast over the input. At locations where the condition is True, the *out* array will be set to the ufunc result. Elsewhere, the *out* array will retain its original value. Note that if an uninitialized *out* array is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.

****kwargs**

For other keyword-only arguments, see the *ufunc docs*.

**Returns**

**y**

[ndarray] y = floor(*x1*/*x2*) This is a scalar if both *x1* and *x2* are scalars.

**See also:**

***`remainder`***

Remainder complementary to floor_divide.

***`divmod`***

Simultaneous floor division and remainder.

***`divide`***

Standard division.

***`floor`***

Round a number to the nearest integer toward minus infinity.

***`ceil`***

Round a number to the nearest integer toward infinity.

### Examples

```
>>> np.floor_divide(7,3)
2
>>> np.floor_divide([1., 2., 3., 4.], 2.5)
array([ 0.,  0.,  1.,  1.])
```

`numpy.`**`float_power`**(*x1*, *x2*, */*, *out=None*, ***, *where=True*, *casting='same_kind'*, *order='K'*, *dtype=None*, *subok=True*[, *signature*, *extobj* ]) **`= <ufunc 'float_power'>`**

First array elements raised to powers from second array, element-wise.

Raise each base in *x1* to the positionally-corresponding power in *x2*. *x1* and *x2* must be broadcastable to the same shape. This differs from the power function in that integers, float16, and float32 are promoted to floats with a minimum precision of float64 so that the result is always inexact. The intent is that the function will return a usable result for negative powers and seldom overflow for positive powers.

New in version 1.12.0.

**Parameters**

**x1**

[array_like] The bases.

**x2**

[array_like] The exponents. If `x1.shape != x2.shape`, they must be broadcastable to a common shape (which becomes the shape of the output).

**out**

[ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

**where**

[array_like, optional] This condition is broadcast over the input. At locations where the condition is True, the *out* array will be set to the ufunc result. Elsewhere, the *out* array will retain its original value. Note that if an uninitialized *out* array is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.

****kwargs**

For other keyword-only arguments, see the *ufunc docs*.

**Returns**

**y**

[ndarray] The bases in *x1* raised to the exponents in *x2*. This is a scalar if both *x1* and *x2* are scalars.

**See also:**

***power***

power function that preserves type

## Examples

Cube each element in a list.

```
>>> x1 = range(6)
>>> x1
[0, 1, 2, 3, 4, 5]
>>> np.float_power(x1, 3)
array([   0.,    1.,    8.,   27.,   64.,  125.])
```

Raise the bases to different exponents.

```
>>> x2 = [1.0, 2.0, 3.0, 3.0, 2.0, 1.0]
>>> np.float_power(x1, x2)
array([  0.,   1.,   8.,  27.,  16.,   5.])
```

The effect of broadcasting.

```
>>> x2 = np.array([[1, 2, 3, 3, 2, 1], [1, 2, 3, 3, 2, 1]])
>>> x2
array([[1, 2, 3, 3, 2, 1],
       [1, 2, 3, 3, 2, 1]])
>>> np.float_power(x1, x2)
array([[  0.,   1.,   8.,  27.,  16.,   5.],
       [  0.,   1.,   8.,  27.,  16.,   5.]])
```

`numpy.`**`fmod`**(*x1*, *x2*, */*, *out=None*, ***, *where=True*, *casting='same_kind'*, *order='K'*, *dtype=None*, *subok=True*[, *signature*, *extobj*]) **`= <ufunc 'fmod'>`**

Return the element-wise remainder of division.

This is the NumPy implementation of the C library function fmod, the remainder has the same sign as the dividend *x1*. It is equivalent to the Matlab(TM) `rem` function and should not be confused with the Python modulus operator `x1 % x2`.

**Parameters**

**x1**

[array_like] Dividend.

**x2**

[array_like] Divisor. If `x1.shape != x2.shape`, they must be broadcastable to a common shape (which becomes the shape of the output).

**out**

[ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

**where**

[array_like, optional] This condition is broadcast over the input. At locations where the condition is True, the *out* array will be set to the ufunc result. Elsewhere, the *out* array will retain its original value. Note that if an uninitialized *out* array is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.

****kwargs**

For other keyword-only arguments, see the *ufunc docs*.

**Returns**

**y**

[array_like] The remainder of the division of *x1* by *x2*. This is a scalar if both *x1* and *x2* are scalars.

**See also:**

**`remainder`**

Equivalent to the Python `%` operator.

`divide`

## Notes

The result of the modulo operation for negative dividend and divisors is bound by conventions. For `fmod`, the sign of result is the sign of the dividend, while for `remainder` the sign of the result is the sign of the divisor. The `fmod` function is equivalent to the Matlab(TM) `rem` function.

### Examples

```
>>> np.fmod([-3, -2, -1, 1, 2, 3], 2)
array([-1,  0, -1,  1,  0,  1])
>>> np.remainder([-3, -2, -1, 1, 2, 3], 2)
array([1, 0, 1, 1, 0, 1])
```

```
>>> np.fmod([5, 3], [2, 2.])
array([ 1.,  1.])
>>> a = np.arange(-3, 3).reshape(3, 2)
>>> a
array([[-3, -2],
       [-1,  0],
       [ 1,  2]])
>>> np.fmod(a, [2,2])
array([[-1,  0],
       [-1,  0],
       [ 1,  0]])
```

`numpy.`**`mod`**(*x1*, *x2*, */*, *out=None*, ***, *where=True*, *casting='same_kind'*, *order='K'*, *dtype=None*, *subok=True*[, *signature*, *extobj* ]) **= <ufunc 'remainder'>**
Return element-wise remainder of division.

Computes the remainder complementary to the *`floor_divide`* function. It is equivalent to the Python modulus operator“x1 % x2“ and has the same sign as the divisor *x2*. The MATLAB function equivalent to `np.remainder` is `mod`.

**Warning:** This should not be confused with:

- Python 3.7's `math.remainder` and C's `remainder`, which computes the IEEE remainder, which are the complement to `round(x1 / x2)`.
- The MATLAB `rem` function and or the C `%` operator which is the complement to `int(x1 / x2)`.

**Parameters**

**x1**

[array_like] Dividend array.

**x2**

[array_like] Divisor array. If `x1.shape != x2.shape`, they must be broadcastable to a common shape (which becomes the shape of the output).

**out**

[ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

**where**

[array_like, optional] This condition is broadcast over the input. At locations where the condition is True, the *out* array will be set to the ufunc result. Elsewhere, the *out* array will retain its original value. Note that if an uninitialized *out* array is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.

****kwargs**

For other keyword-only arguments, see the *ufunc docs*.

**Returns**

**y**

[ndarray] The element-wise remainder of the quotient `floor_divide(x1, x2)`. This is a scalar if both *x1* and *x2* are scalars.

**See also:**

**`floor_divide`**

Equivalent of Python `//` operator.

**`divmod`**

Simultaneous floor division and remainder.

**`fmod`**

Equivalent of the MATLAB `rem` function.

`divide`, `floor`

## Notes

Returns 0 when *x2* is 0 and both *x1* and *x2* are (arrays of) integers. `mod` is an alias of `remainder`.

## Examples

```
>>> np.remainder([4, 7], [2, 3])
array([0, 1])
>>> np.remainder(np.arange(7), 5)
array([0, 1, 2, 3, 4, 0, 1])
```

`numpy.`**`modf`**(*x*[, *out1*, *out2* ], /[, *out=(None, None)* ], *, *where=True*, *casting='same_kind'*, *order='K'*, *dtype=None*, *subok=True*[, *signature*, *extobj* ]) **`= <ufunc 'modf'>`**

Return the fractional and integral parts of an array, element-wise.

The fractional and integral parts are negative if the given number is negative.

**Parameters**

**x**

[array_like] Input array.

**out**

[ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

**where**

[array_like, optional] This condition is broadcast over the input. At locations where the condition is True, the *out* array will be set to the ufunc result. Elsewhere, the *out* array will retain its original value. Note that if an uninitialized *out* array is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.

****kwargs**

For other keyword-only arguments, see the *ufunc docs*.

**Returns**

**y1**

[ndarray] Fractional part of *x*. This is a scalar if *x* is a scalar.

**y2**

[ndarray] Integral part of *x*. This is a scalar if *x* is a scalar.

**See also:**

***`divmod`***

`divmod(x, 1)` is equivalent to `modf` with the return values switched, except it always has a positive remainder.

## Notes

For integer input the return values are floats.

## Examples

```
>>> np.modf([0, 3.5])
(array([ 0. ,  0.5]), array([ 0.,  3.]))
>>> np.modf(-0.5)
(-0.5, -0)
```

`numpy.`**`remainder`**(*x1*, *x2*, /, *out=None*, *, *where=True*, *casting='same_kind'*, *order='K'*, *dtype=None*, *subok=True*[, *signature*, *extobj* ]) **`= <ufunc 'remainder'>`**

Return element-wise remainder of division.

Computes the remainder complementary to the *`floor_divide`* function. It is equivalent to the Python modulus operator“x1 % x2“ and has the same sign as the divisor *x2*. The MATLAB function equivalent to `np.remainder` is `mod`.

**Warning:** This should not be confused with:

- Python 3.7's `math.remainder` and C's `remainder`, which computes the IEEE remainder, which are the complement to `round(x1 / x2)`.
- The MATLAB `rem` function and or the C `%` operator which is the complement to `int(x1 / x2)`.

**Parameters**

**x1**

[array_like] Dividend array.

**x2**

[array_like] Divisor array. If `x1.shape != x2.shape`, they must be broadcastable to a common shape (which becomes the shape of the output).

**out**

[ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

**where**

[array_like, optional] This condition is broadcast over the input. At locations where the condition is True, the *out* array will be set to the ufunc result. Elsewhere, the *out* array will retain its original value. Note that if an uninitialized *out* array is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.

****kwargs**

For other keyword-only arguments, see the *ufunc docs*.

**Returns**

**y**

[ndarray] The element-wise remainder of the quotient `floor_divide(x1, x2)`. This is a scalar if both *x1* and *x2* are scalars.

**See also:**

***`floor_divide`***

Equivalent of Python `//` operator.

***`divmod`***

Simultaneous floor division and remainder.

***`fmod`***

Equivalent of the MATLAB `rem` function.

`divide`, `floor`

## Notes

Returns 0 when *x2* is 0 and both *x1* and *x2* are (arrays of) integers. `mod` is an alias of `remainder`.

## Examples

```
>>> np.remainder([4, 7], [2, 3])
array([0, 1])
>>> np.remainder(np.arange(7), 5)
array([0, 1, 2, 3, 4, 0, 1])
```

`numpy.`**`divmod`**(*x1*, *x2*[, *out1*, *out2* ], /[, *out=(None, None)* ], *, *where=True*, *casting='same_kind'*, *order='K'*, *dtype=None*, *subok=True*[, *signature*, *extobj* ]) **`= <ufunc 'divmod'>`**

Return element-wise quotient and remainder simultaneously.

New in version 1.13.0.

`np.divmod(x, y)` is equivalent to `(x // y, x % y)`, but faster because it avoids redundant work. It is used to implement the Python built-in function `divmod` on NumPy arrays.

**Parameters**

**x1**

[array_like] Dividend array.

**x2**

[array_like] Divisor array. If `x1.shape != x2.shape`, they must be broadcastable to a common shape (which becomes the shape of the output).

**out**

[ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

**where**

[array_like, optional] This condition is broadcast over the input. At locations where the condition is True, the *out* array will be set to the ufunc result. Elsewhere, the *out* array will retain its original value. Note that if an uninitialized *out* array is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.

****kwargs**

For other keyword-only arguments, see the *ufunc docs*.

**Returns**

**out1**

[ndarray] Element-wise quotient resulting from floor division. This is a scalar if both *x1* and *x2* are scalars.

**out2**

[ndarray] Element-wise remainder from floor division. This is a scalar if both *x1* and *x2* are scalars.

**See also:**

***`floor_divide`***

Equivalent to Python's `//` operator.

*remainder*

Equivalent to Python's `%` operator.

*modf*

Equivalent to `divmod(x, 1)` for positive `x` with the return values switched.

### Examples

```
>>> np.divmod(np.arange(5), 3)
(array([0, 0, 0, 1, 1]), array([0, 1, 2, 0, 1]))
```

## 4.19.10 Handling complex numbers

| | |
|---|---|
| *angle*(z[, deg]) | Return the angle of the complex argument. |
| *real*(val) | Return the real part of the complex argument. |
| *imag*(val) | Return the imaginary part of the complex argument. |
| *conj*(x, /[, out, where, casting, order, …]) | Return the complex conjugate, element-wise. |
| *conjugate*(x, /[, out, where, casting, …]) | Return the complex conjugate, element-wise. |

numpy.**angle**(*z*, *deg=False*)

Return the angle of the complex argument.

**Parameters**

**z**

[array_like] A complex number or sequence of complex numbers.

**deg**

[bool, optional] Return angle in degrees if True, radians if False (default).

**Returns**

**angle**

[ndarray or scalar] The counterclockwise angle from the positive real axis on the complex plane in the range `(-pi, pi]`, with dtype as numpy.float64.

**..versionchanged:: 1.16.0**

This function works on subclasses of ndarray like *ma.array*.

**See also:**

*arctan2*, *absolute*

### Notes

Although the angle of the complex number 0 is undefined, `numpy.angle(0)` returns the value 0.

### Examples

```
>>> np.angle([1.0, 1.0j, 1+1j])               # in radians
array([ 0.        ,  1.57079633,  0.78539816]) # may vary
>>> np.angle(1+1j, deg=True)                  # in degrees
45.0
```

`numpy.`**`real`**`(`*val*`)`

Return the real part of the complex argument.

**Parameters**

**val**

[array_like] Input array.

**Returns**

**out**

[ndarray or scalar] The real component of the complex argument. If *val* is real, the type of *val* is used for the output. If *val* has complex elements, the returned type is float.

**See also:**

*`real_if_close`*, *`imag`*, *`angle`*

### Examples

```
>>> a = np.array([1+2j, 3+4j, 5+6j])
>>> a.real
array([1.,  3.,  5.])
>>> a.real = 9
>>> a
array([9.+2.j,  9.+4.j,  9.+6.j])
>>> a.real = np.array([9, 8, 7])
>>> a
array([9.+2.j,  8.+4.j,  7.+6.j])
>>> np.real(1 + 1j)
1.0
```

`numpy.`**`imag`**`(`*val*`)`

Return the imaginary part of the complex argument.

**Parameters**

**val**

[array_like] Input array.

**Returns**

**out**

[ndarray or scalar] The imaginary component of the complex argument. If *val* is real, the type of *val* is used for the output. If *val* has complex elements, the returned type is float.

**See also:**

*real*, *angle*, *real_if_close*

### Examples

```
>>> a = np.array([1+2j, 3+4j, 5+6j])
>>> a.imag
array([2.,  4.,  6.])
>>> a.imag = np.array([8, 10, 12])
>>> a
array([1. +8.j,  3.+10.j,  5.+12.j])
>>> np.imag(1 + 1j)
1.0
```

`numpy.`**`conj`**(*x*, */*, *out=None*, ***, *where=True*, *casting='same_kind'*, *order='K'*, *dtype=None*, *subok=True*[, *signature*, *extobj*]) **`= <ufunc 'conjugate'>`**

Return the complex conjugate, element-wise.

The complex conjugate of a complex number is obtained by changing the sign of its imaginary part.

**Parameters**

**x**

[array_like] Input value.

**out**

[ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

**where**

[array_like, optional] This condition is broadcast over the input. At locations where the condition is True, the *out* array will be set to the ufunc result. Elsewhere, the *out* array will retain its original value. Note that if an uninitialized *out* array is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.

****kwargs**

For other keyword-only arguments, see the *ufunc docs*.

**Returns**

**y**

[ndarray] The complex conjugate of *x*, with same dtype as *y*. This is a scalar if *x* is a scalar.

### Notes

*`conj`* is an alias for *`conjugate`*:

```
>>> np.conj is np.conjugate
True
```

### Examples

```
>>> np.conjugate(1+2j)
(1-2j)
```

```
>>> x = np.eye(2) + 1j * np.eye(2)
>>> np.conjugate(x)
array([[ 1.-1.j,  0.-0.j],
       [ 0.-0.j,  1.-1.j]])
```

`numpy.`**`conjugate`**(*x*, */*, *out=None*, ***, *where=True*, *casting='same_kind'*, *order='K'*, *dtype=None*, *subok=True*[, *signature*, *extobj* ]) **`= <ufunc 'conjugate'>`**

Return the complex conjugate, element-wise.

The complex conjugate of a complex number is obtained by changing the sign of its imaginary part.

**Parameters**

**x**

[array_like] Input value.

**out**

[ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

**where**

[array_like, optional] This condition is broadcast over the input. At locations where the condition is True, the *out* array will be set to the ufunc result. Elsewhere, the *out* array will retain its original value. Note that if an uninitialized *out* array is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.

****kwargs**

For other keyword-only arguments, see the *ufunc docs*.

**Returns**

**y**

[ndarray] The complex conjugate of *x*, with same dtype as *y*. This is a scalar if *x* is a scalar.

### Notes

`conj` is an alias for `conjugate`:

```
>>> np.conj is np.conjugate
True
```

### Examples

```
>>> np.conjugate(1+2j)
(1-2j)
```

```
>>> x = np.eye(2) + 1j * np.eye(2)
>>> np.conjugate(x)
array([[ 1.-1.j,  0.-0.j],
       [ 0.-0.j,  1.-1.j]])
```

## 4.19.11 Miscellaneous

| | |
|---|---|
| `convolve`(a, v[, mode]) | Returns the discrete, linear convolution of two one-dimensional sequences. |
| `clip`(a, a_min, a_max[, out]) | Clip (limit) the values in an array. |
| `sqrt`(x, /[, out, where, casting, order, …]) | Return the non-negative square-root of an array, element-wise. |
| `cbrt`(x, /[, out, where, casting, order, …]) | Return the cube-root of an array, element-wise. |
| `square`(x, /[, out, where, casting, order, …]) | Return the element-wise square of the input. |
| `absolute`(x, /[, out, where, casting, order, …]) | Calculate the absolute value element-wise. |
| `fabs`(x, /[, out, where, casting, order, …]) | Compute the absolute values element-wise. |
| `sign`(x, /[, out, where, casting, order, …]) | Returns an element-wise indication of the sign of a number. |
| `heaviside`(x1, x2, /[, out, where, casting, …]) | Compute the Heaviside step function. |
| `maximum`(x1, x2, /[, out, where, casting, …]) | Element-wise maximum of array elements. |
| `minimum`(x1, x2, /[, out, where, casting, …]) | Element-wise minimum of array elements. |
| `fmax`(x1, x2, /[, out, where, casting, …]) | Element-wise maximum of array elements. |
| `fmin`(x1, x2, /[, out, where, casting, …]) | Element-wise minimum of array elements. |
| `nan_to_num`(x[, copy, nan, posinf, neginf]) | Replace NaN with zero and infinity with large finite numbers (default behaviour) or with the numbers defined by the user using the `nan`, *posinf* and/or *neginf* keywords. |
| `real_if_close`(a[, tol]) | If input is complex with all imaginary parts close to zero, return real parts. |
| `interp`(x, xp, fp[, left, right, period]) | One-dimensional linear interpolation. |

`numpy.`**`convolve`**(*a*, *v*, *mode='full'*)

Returns the discrete, linear convolution of two one-dimensional sequences.

The convolution operator is often seen in signal processing, where it models the effect of a linear time-invariant system on a signal [1]. In probability theory, the sum of two independent random variables is distributed according to the convolution of their individual distributions.

If *v* is longer than *a*, the arrays are swapped before computation.

**Parameters**

**a**

[(N,) array_like] First one-dimensional input array.

**v**

[(M,) array_like] Second one-dimensional input array.

**mode**

[{'full', 'valid', 'same'}, optional]

**'full':**

By default, mode is 'full'. This returns the convolution at each point of overlap, with an output shape of (N+M-1,). At the end-points of the convolution, the signals do not overlap completely, and boundary effects may be seen.

**'same':**

Mode 'same' returns output of length `max(M, N)`. Boundary effects are still visible.

**'valid':**

Mode 'valid' returns output of length `max(M, N) - min(M, N) + 1`. The convolution product is only given for points where the signals overlap completely. Values outside the signal boundary have no effect.

**Returns**

**out**

[ndarray] Discrete, linear convolution of *a* and *v*.

**See also:**

**`scipy.signal.fftconvolve`**

Convolve two arrays using the Fast Fourier Transform.

**`scipy.linalg.toeplitz`**

Used to construct the convolution operator.

***`polymul`***

Polynomial multiplication. Same output as convolve, but also accepts poly1d objects as input.

## Notes

The discrete convolution operation is defined as

$$(a * v)[n] = \sum_{m=-\infty}^{\infty} a[m]v[n-m]$$

It can be shown that a convolution $x(t) * y(t)$ in time/space is equivalent to the multiplication $X(f)Y(f)$ in the Fourier domain, after appropriate padding (padding is necessary to prevent circular convolution). Since multiplication is more efficient (faster) than convolution, the function `scipy.signal.fftconvolve` exploits the FFT to calculate the convolution of large data-sets.

### References

[1]

### Examples

Note how the convolution operator flips the second array before "sliding" the two across one another:

```
>>> np.convolve([1, 2, 3], [0, 1, 0.5])
array([0. , 1. , 2.5, 4. , 1.5])
```

Only return the middle values of the convolution. Contains boundary effects, where zeros are taken into account:

```
>>> np.convolve([1,2,3],[0,1,0.5], 'same')
array([1. ,  2.5,  4. ])
```

The two arrays are of the same length, so there is only one position where they completely overlap:

```
>>> np.convolve([1,2,3],[0,1,0.5], 'valid')
array([2.5])
```

`numpy.`**`clip`**(*a*, *a_min*, *a_max*, *out=None*, ***kwargs*)

Clip (limit) the values in an array.

Given an interval, values outside the interval are clipped to the interval edges. For example, if an interval of `[0, 1]` is specified, values smaller than 0 become 0, and values larger than 1 become 1.

Equivalent to but faster than `np.minimum(a_max, np.maximum(a, a_min))`.

No check is performed to ensure `a_min < a_max`.

**Parameters**

**a**

[array_like] Array containing elements to clip.

**a_min**

[scalar or array_like or None] Minimum value. If None, clipping is not performed on lower interval edge. Not more than one of *a_min* and *a_max* may be None.

**a_max**

[scalar or array_like or None] Maximum value. If None, clipping is not performed on upper interval edge. Not more than one of *a_min* and *a_max* may be None. If *a_min* or *a_max* are array_like, then the three arrays will be broadcasted to match their shapes.

**out**

[ndarray, optional] The results will be placed in this array. It may be the input array for in-place clipping. *out* must be of the right shape to hold the output. Its type is preserved.

****kwargs**

For other keyword-only arguments, see the *ufunc docs*.

New in version 1.17.0.

**Returns**

**clipped_array**

[ndarray] An array with the elements of *a*, but where values < *a_min* are replaced with *a_min*, and those > *a_max* with *a_max*.

**See also:**

`ufuncs-output-type`

## Examples

```
>>> a = np.arange(10)
>>> np.clip(a, 1, 8)
array([1, 1, 2, 3, 4, 5, 6, 7, 8, 8])
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> np.clip(a, 3, 6, out=a)
array([3, 3, 3, 3, 4, 5, 6, 6, 6, 6])
>>> a = np.arange(10)
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> np.clip(a, [3, 4, 1, 1, 1, 4, 4, 4, 4, 4], 8)
array([3, 4, 2, 3, 4, 5, 6, 7, 8, 8])
```

`numpy.`**`sqrt`** (*x*, */*, *out=None*, ***, *where=True*, *casting='same_kind'*, *order='K'*, *dtype=None*, *subok=True*[, *signature*, *extobj* ]) **`= <ufunc 'sqrt'>`**

Return the non-negative square-root of an array, element-wise.

**Parameters**

**x**

[array_like] The values whose square-roots are required.

**out**

[ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

**where**

[array_like, optional] This condition is broadcast over the input. At locations where the condition is True, the *out* array will be set to the ufunc result. Elsewhere, the *out* array will retain its original value. Note that if an uninitialized *out* array is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.

****kwargs**

For other keyword-only arguments, see the *ufunc docs*.

**Returns**

**y**

[ndarray] An array of the same shape as *x*, containing the positive square-root of each element in *x*. If any element in *x* is complex, a complex array is returned (and the square-roots of negative reals are calculated). If all of the elements in *x* are real, so is *y*, with negative elements returning `nan`. If *out* was provided, *y* is a reference to it. This is a scalar if *x* is a scalar.

**See also:**

**`lib.scimath.sqrt`**
A version which returns complex numbers when given negative reals.

## Notes

*sqrt* has–consistent with common convention–as its branch cut the real "interval" [*-inf*, 0), and is continuous from above on it. A branch cut is a curve in the complex plane across which a given complex function fails to be continuous.

## Examples

```
>>> np.sqrt([1,4,9])
array([ 1.,  2.,  3.])
```

```
>>> np.sqrt([4, -1, -3+4J])
array([ 2.+0.j,  0.+1.j,  1.+2.j])
```

```
>>> np.sqrt([4, -1, np.inf])
array([ 2., nan, inf])
```

`numpy.`**`cbrt`** (*x*, */*, *out=None*, ***, *where=True*, *casting='same_kind'*, *order='K'*, *dtype=None*, *subok=True*[, *signature*, *extobj* ]) **`= <ufunc 'cbrt'>`**

Return the cube-root of an array, element-wise.

New in version 1.10.0.

**Parameters**

**x**

[array_like] The values whose cube-roots are required.

**out**

[ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

**where**

[array_like, optional] This condition is broadcast over the input. At locations where the condition is True, the *out* array will be set to the ufunc result. Elsewhere, the *out* array will retain its original value. Note that if an uninitialized *out* array is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.

****kwargs**

For other keyword-only arguments, see the *ufunc docs*.

**Returns**

**y**

[ndarray] An array of the same shape as *x*, containing the cube cube-root of each element in *x*. If *out* was provided, *y* is a reference to it. This is a scalar if *x* is a scalar.

### Examples

```
>>> np.cbrt([1,8,27])
array([ 1.,  2.,  3.])
```

`numpy.`**`square`**(*x*, */*, *out=None*, ***, *where=True*, *casting='same_kind'*, *order='K'*, *dtype=None*, *subok=True*[, *signature*, *extobj* ]) **`= <ufunc 'square'>`**

Return the element-wise square of the input.

**Parameters**

**x**

[array_like] Input data.

**out**

[ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

**where**

[array_like, optional] This condition is broadcast over the input. At locations where the condition is True, the *out* array will be set to the ufunc result. Elsewhere, the *out* array will retain its original value. Note that if an uninitialized *out* array is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.

****kwargs**

For other keyword-only arguments, see the *ufunc docs*.

**Returns**

**out**

[ndarray or scalar] Element-wise *x*x*, of the same shape and dtype as *x*. This is a scalar if *x* is a scalar.

**See also:**

*`numpy.linalg.matrix_power`*, *`sqrt`*, *`power`*

## Examples

```
>>> np.square([-1j, 1])
array([-1.-0.j,  1.+0.j])
```

`numpy.`**`absolute`**(*x*, */*, *out=None*, ***, *where=True*, *casting='same_kind'*, *order='K'*, *dtype=None*, *subok=True*[, *signature*, *extobj*]) **`= <ufunc 'absolute'>`**

Calculate the absolute value element-wise.

`np.abs` is a shorthand for this function.

**Parameters**

**x**

[array_like] Input array.

**out**

[ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

**where**

[array_like, optional] This condition is broadcast over the input. At locations where the condition is True, the *out* array will be set to the ufunc result. Elsewhere, the *out* array will retain its original value. Note that if an uninitialized *out* array is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.

****kwargs**

For other keyword-only arguments, see the *ufunc docs*.

**Returns**

**absolute**

[ndarray] An ndarray containing the absolute value of each element in *x*. For complex input, `a + ib`, the absolute value is $\sqrt{a^2 + b^2}$. This is a scalar if *x* is a scalar.

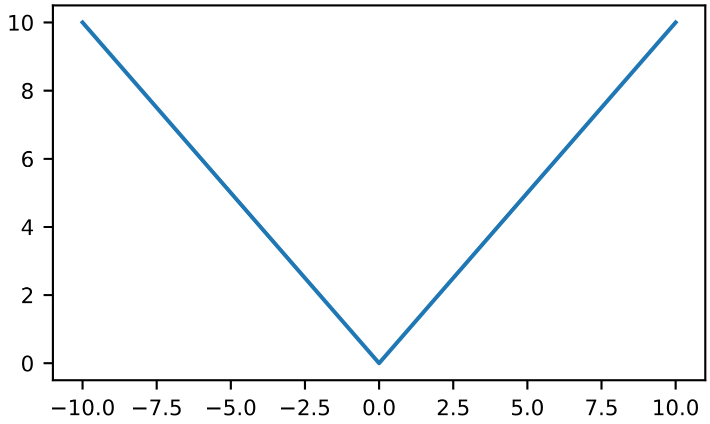
## Examples

```
>>> x = np.array([-1.2, 1.2])
>>> np.absolute(x)
array([ 1.2,  1.2])
>>> np.absolute(1.2 + 1j)
1.5620499351813308
```

Plot the function over `[-10, 10]`:

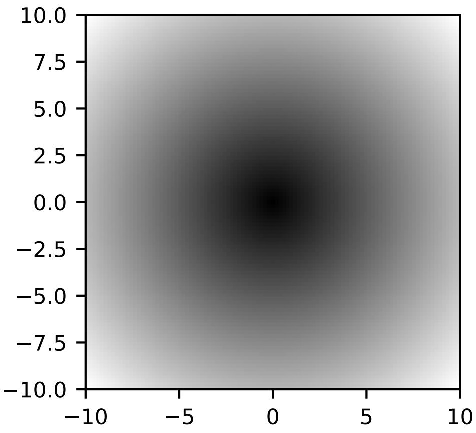
```
>>> import matplotlib.pyplot as plt
```

```
>>> x = np.linspace(start=-10, stop=10, num=101)
>>> plt.plot(x, np.absolute(x))
>>> plt.show()
```



Plot the function over the complex plane:

```
>>> xx = x + 1j * x[:, np.newaxis]
>>> plt.imshow(np.abs(xx), extent=[-10, 10, -10, 10], cmap='gray')
>>> plt.show()
```



`numpy.`**`fabs`** (*x*, */*, *out=None*, ***, *where=True*, *casting='same_kind'*, *order='K'*, *dtype=None*, *subok=True*[, *signature*, *extobj*]) **`= <ufunc 'fabs'>`**

Compute the absolute values element-wise.

This function returns the absolute values (positive magnitude) of the data in *x*. Complex values are not handled, use `absolute` to find the absolute values of complex data.

**Parameters**

**x**

[array_like] The array of numbers for which the absolute values are required. If *x* is a scalar, the result *y* will also be a scalar.

**out**

[ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

**where**

[array_like, optional] This condition is broadcast over the input. At locations where the condition is True, the *out* array will be set to the ufunc result. Elsewhere, the *out* array will retain its original value. Note that if an uninitialized *out* array is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.

****kwargs**

For other keyword-only arguments, see the *ufunc docs*.

**Returns**

**y**

[ndarray or scalar] The absolute values of *x*, the returned values are always floats. This is a scalar if *x* is a scalar.

**See also:**

***`absolute`***

Absolute values including `complex` types.

## Examples

```
>>> np.fabs(-1)
1.0
>>> np.fabs([-1.2, 1.2])
array([ 1.2,  1.2])
```

`numpy.`**`sign`**(*x*, */*, *out=None*, ***, *where=True*, *casting='same_kind'*, *order='K'*, *dtype=None*, *subok=True*[, *signature*, *extobj* ]) **`= <ufunc 'sign'>`**

Returns an element-wise indication of the sign of a number.

The *`sign`* function returns `-1 if x < 0, 0 if x==0, 1 if x > 0`. nan is returned for nan inputs.

For complex inputs, the *`sign`* function returns `sign(x.real) + 0j if x.real != 0 else sign(x.imag) + 0j`.

complex(nan, 0) is returned for complex nan inputs.

**Parameters**

**x**

[array_like] Input values.

**out**

[ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

**where**

[array_like, optional] This condition is broadcast over the input. At locations where the condition is True, the *out* array will be set to the ufunc result. Elsewhere, the *out* array will retain its original value. Note that if an uninitialized *out* array is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.

****kwargs**

For other keyword-only arguments, see the *ufunc docs*.

**Returns**

**y**

[ndarray] The sign of *x*. This is a scalar if *x* is a scalar.

## Notes

There is more than one definition of sign in common use for complex numbers. The definition used here is equivalent to $x/\sqrt{x * x}$ which is different from a common alternative, $x/|x|$.

## Examples

```
>>> np.sign([-5., 4.5])
array([-1.,  1.])
>>> np.sign(0)
0
>>> np.sign(5-2j)
(1+0j)
```

`numpy.`**`heaviside`**(*x1*, *x2*, */*, *out=None*, ***, *where=True*, *casting='same_kind'*, *order='K'*, *dtype=None*, *subok=True*[, *signature*, *extobj*]) **`= <ufunc 'heaviside'>`**

Compute the Heaviside step function.

The Heaviside step function is defined as:

```
                      0   if x1 < 0
heaviside(x1, x2) =  x2   if x1 == 0
                      1   if x1 > 0
```

where *x2* is often taken to be 0.5, but 0 and 1 are also sometimes used.

**Parameters**

**x1**

[array_like] Input values.

**x2**

[array_like] The value of the function when x1 is 0. If `x1.shape != x2.shape`, they must be broadcastable to a common shape (which becomes the shape of the output).

**out**

[ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

**where**

[array_like, optional] This condition is broadcast over the input. At locations where the condition is True, the *out* array will be set to the ufunc result. Elsewhere, the *out* array will retain its original value. Note that if an uninitialized *out* array is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.

****kwargs**

For other keyword-only arguments, see the *ufunc docs*.

**Returns**

**out**

[ndarray or scalar] The output array, element-wise Heaviside step function of *x1*. This is a scalar if both *x1* and *x2* are scalars.

## Notes

New in version 1.13.0.

## References

## Examples

```
>>> np.heaviside([-1.5, 0, 2.0], 0.5)
array([ 0. ,  0.5,  1. ])
>>> np.heaviside([-1.5, 0, 2.0], 1)
array([ 0.,  1.,  1.])
```

`numpy.`**`maximum`**(*x1*, *x2*, */*, *out=None*, ***, *where=True*, *casting='same_kind'*, *order='K'*, *dtype=None*, *subok=True*[, *signature*, *extobj* ]) **`= <ufunc 'maximum'>`**

Element-wise maximum of array elements.

Compare two arrays and returns a new array containing the element-wise maxima. If one of the elements being compared is a NaN, then that element is returned. If both elements are NaNs then the first is returned. The latter distinction is important for complex NaNs, which are defined as at least one of the real or imaginary parts being a NaN. The net effect is that NaNs are propagated.

**Parameters**

**x1, x2**

[array_like] The arrays holding the elements to be compared. If `x1.shape != x2.shape`, they must be broadcastable to a common shape (which becomes the shape of the output).

**out**

[ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

**where**

[array_like, optional] This condition is broadcast over the input. At locations where the condition is True, the *out* array will be set to the ufunc result. Elsewhere, the *out* array will retain its original value. Note that if an uninitialized *out* array is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.

****kwargs**

For other keyword-only arguments, see the *ufunc docs*.

**Returns**

**y**

[ndarray or scalar] The maximum of *x1* and *x2*, element-wise. This is a scalar if both *x1* and *x2* are scalars.

**See also:**

***`minimum`***

Element-wise minimum of two arrays, propagates NaNs.

***`fmax`***

Element-wise maximum of two arrays, ignores NaNs.

***`amax`***

The maximum value of an array along a given axis, propagates NaNs.

***`nanmax`***

The maximum value of an array along a given axis, ignores NaNs.

*`fmin`*, *`amin`*, *`nanmin`*

## Notes

The maximum is equivalent to `np.where(x1 >= x2, x1, x2)` when neither x1 nor x2 are nans, but it is faster and does proper broadcasting.

### Examples

```
>>> np.maximum([2, 3, 4], [1, 5, 2])
array([2, 5, 4])
```

```
>>> np.maximum(np.eye(2), [0.5, 2]) # broadcasting
array([[ 1. ,  2. ],
       [ 0.5,  2. ]])
```

```
>>> np.maximum([np.nan, 0, np.nan], [0, np.nan, np.nan])
array([nan, nan, nan])
>>> np.maximum(np.Inf, 1)
inf
```

`numpy.`**`minimum`**(*x1*, *x2*, /, *out=None*, *, *where=True*, *casting='same_kind'*, *order='K'*, *dtype=None*, *subok=True*[, *signature*, *extobj*]) **`= <ufunc 'minimum'>`**

Element-wise minimum of array elements.

Compare two arrays and returns a new array containing the element-wise minima. If one of the elements being compared is a NaN, then that element is returned. If both elements are NaNs then the first is returned. The latter distinction is important for complex NaNs, which are defined as at least one of the real or imaginary parts being a NaN. The net effect is that NaNs are propagated.

**Parameters**

**x1, x2**

[array_like] The arrays holding the elements to be compared. If `x1.shape != x2.shape`, they must be broadcastable to a common shape (which becomes the shape of the output).

**out**

[ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

**where**

[array_like, optional] This condition is broadcast over the input. At locations where the condition is True, the *out* array will be set to the ufunc result. Elsewhere, the *out* array will retain its original value. Note that if an uninitialized *out* array is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.

****kwargs**

For other keyword-only arguments, see the *ufunc docs*.

**Returns**

**y**

[ndarray or scalar] The minimum of *x1* and *x2*, element-wise. This is a scalar if both *x1* and *x2* are scalars.

**See also:**

***`maximum`***

Element-wise maximum of two arrays, propagates NaNs.

***`fmin`***

Element-wise minimum of two arrays, ignores NaNs.

***`amin`***

The minimum value of an array along a given axis, propagates NaNs.

***`nanmin`***

The minimum value of an array along a given axis, ignores NaNs.

*`fmax`*, *`amax`*, *`nanmax`*

### Notes

The minimum is equivalent to `np.where(x1 <= x2, x1, x2)` when neither x1 nor x2 are NaNs, but it is faster and does proper broadcasting.

### Examples

```
>>> np.minimum([2, 3, 4], [1, 5, 2])
array([1, 3, 2])
```

```
>>> np.minimum(np.eye(2), [0.5, 2]) # broadcasting
array([[ 0.5,  0. ],
       [ 0. ,  1. ]])
```

```
>>> np.minimum([np.nan, 0, np.nan],[0, np.nan, np.nan])
array([nan, nan, nan])
>>> np.minimum(-np.Inf, 1)
-inf
```

`numpy.`**`fmax`**(*x1*, *x2*, */*, *out=None*, ***, *where=True*, *casting='same_kind'*, *order='K'*, *dtype=None*, *subok=True*[, *signature*, *extobj*]) **`= <ufunc 'fmax'>`**

Element-wise maximum of array elements.

Compare two arrays and returns a new array containing the element-wise maxima. If one of the elements being compared is a NaN, then the non-nan element is returned. If both elements are NaNs then the first is returned. The latter distinction is important for complex NaNs, which are defined as at least one of the real or imaginary parts being a NaN. The net effect is that NaNs are ignored when possible.

**Parameters**

**x1, x2**

[array_like] The arrays holding the elements to be compared. If `x1.shape != x2.shape`, they must be broadcastable to a common shape (which becomes the shape of the output).

**out**

[ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

**where**

[array_like, optional] This condition is broadcast over the input. At locations where the condition is True, the *out* array will be set to the ufunc result. Elsewhere, the *out* array will retain its original value. Note that if an uninitialized *out* array is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.

****kwargs**

For other keyword-only arguments, see the *ufunc docs*.

**Returns**

**y**

[ndarray or scalar] The maximum of *x1* and *x2*, element-wise. This is a scalar if both *x1* and *x2* are scalars.

**See also:**

***`fmin`***

Element-wise minimum of two arrays, ignores NaNs.

***`maximum`***

Element-wise maximum of two arrays, propagates NaNs.

***`amax`***

The maximum value of an array along a given axis, propagates NaNs.

***`nanmax`***

The maximum value of an array along a given axis, ignores NaNs.

`minimum`, `amin`, `nanmin`

## Notes

New in version 1.3.0.

The fmax is equivalent to `np.where(x1 >= x2, x1, x2)` when neither x1 nor x2 are NaNs, but it is faster and does proper broadcasting.

## Examples

```
>>> np.fmax([2, 3, 4], [1, 5, 2])
array([ 2.,  5.,  4.])
```

```
>>> np.fmax(np.eye(2), [0.5, 2])
array([[ 1. ,  2. ],
       [ 0.5,  2. ]])
```

```
>>> np.fmax([np.nan, 0, np.nan],[0, np.nan, np.nan])
array([ 0.,  0., nan])
```

`numpy.`**`fmin`** (*x1*, *x2*, */*, *out=None*, ***, *where=True*, *casting='same_kind'*, *order='K'*, *dtype=None*, *subok=True*[, *signature*, *extobj* ]) **`= <ufunc 'fmin'>`**

Element-wise minimum of array elements.

Compare two arrays and returns a new array containing the element-wise minima. If one of the elements being compared is a NaN, then the non-nan element is returned. If both elements are NaNs then the first is returned. The latter distinction is important for complex NaNs, which are defined as at least one of the real or imaginary parts being a NaN. The net effect is that NaNs are ignored when possible.

**Parameters**

**x1, x2**

[array_like] The arrays holding the elements to be compared. If `x1.shape != x2.shape`, they must be broadcastable to a common shape (which becomes the shape of the output).

**out**

[ndarray, None, or tuple of ndarray and None, optional] A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

**where**

[array_like, optional] This condition is broadcast over the input. At locations where the condition is True, the *out* array will be set to the ufunc result. Elsewhere, the *out* array will retain its original value. Note that if an uninitialized *out* array is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.

****kwargs**

For other keyword-only arguments, see the *ufunc docs*.

**Returns**

**y**

[ndarray or scalar] The minimum of *x1* and *x2*, element-wise. This is a scalar if both *x1* and *x2* are scalars.

**See also:**

**`fmax`**

Element-wise maximum of two arrays, ignores NaNs.

**`minimum`**

Element-wise minimum of two arrays, propagates NaNs.

**`amin`**

The minimum value of an array along a given axis, propagates NaNs.

**`nanmin`**

The minimum value of an array along a given axis, ignores NaNs.

`maximum`, `amax`, `nanmax`

## Notes

New in version 1.3.0.

The fmin is equivalent to `np.where(x1 <= x2, x1, x2)` when neither x1 nor x2 are NaNs, but it is faster and does proper broadcasting.

## Examples

```
>>> np.fmin([2, 3, 4], [1, 5, 2])
array([1, 3, 2])
```

```
>>> np.fmin(np.eye(2), [0.5, 2])
array([[ 0.5,  0. ],
       [ 0. ,  1. ]])
```

```
>>> np.fmin([np.nan, 0, np.nan],[0, np.nan, np.nan])
array([ 0.,  0., nan])
```

`numpy.`**`nan_to_num`**(*x*, *copy=True*, *nan=0.0*, *posinf=None*, *neginf=None*)

Replace NaN with zero and infinity with large finite numbers (default behaviour) or with the numbers defined by the user using the `nan`, *posinf* and/or *neginf* keywords.

If *x* is inexact, NaN is replaced by zero or by the user defined value in `nan` keyword, infinity is replaced by the largest finite floating point values representable by `x.dtype` or by the user defined value in *posinf* keyword and -infinity is replaced by the most negative finite floating point values representable by `x.dtype` or by the user defined value in *neginf* keyword.

For complex dtypes, the above is applied to each of the real and imaginary components of *x* separately.

If *x* is not inexact, then no replacements are made.

**Parameters**

**x**

[scalar or array_like] Input data.

**copy**

[bool, optional] Whether to create a copy of *x* (True) or to replace values in-place (False). The in-place operation only occurs if casting to an array does not require a copy. Default is True.

New in version 1.13.

**nan**

[int, float, optional] Value to be used to fill NaN values. If no value is passed then NaN values will be replaced with 0.0.

New in version 1.17.

**posinf**

[int, float, optional] Value to be used to fill positive infinity values. If no value is passed then positive infinity values will be replaced with a very large number.

New in version 1.17.

**neginf**

[int, float, optional] Value to be used to fill negative infinity values. If no value is passed then negative infinity values will be replaced with a very small (or negative) number.

New in version 1.17.

**Returns**

**out**

[ndarray] *x*, with the non-finite values replaced. If `copy` is False, this may be *x* itself.

**See also:**

**`isinf`**

Shows which elements are positive or negative infinity.

**`isneginf`**

Shows which elements are negative infinity.

**`isposinf`**

Shows which elements are positive infinity.

**`isnan`**

Shows which elements are Not a Number (NaN).

**`isfinite`**

Shows which elements are finite (not NaN, not infinity)

## Notes

NumPy uses the IEEE Standard for Binary Floating-Point for Arithmetic (IEEE 754). This means that Not a Number is not equivalent to infinity.

## Examples

```
>>> np.nan_to_num(np.inf)
1.7976931348623157e+308
>>> np.nan_to_num(-np.inf)
-1.7976931348623157e+308
>>> np.nan_to_num(np.nan)
0.0
>>> x = np.array([np.inf, -np.inf, np.nan, -128, 128])
>>> np.nan_to_num(x)
array([ 1.79769313e+308, -1.79769313e+308,  0.00000000e+000, # may vary
       -1.28000000e+002,  1.28000000e+002])
>>> np.nan_to_num(x, nan=-9999, posinf=33333333, neginf=33333333)
array([ 3.3333333e+07,  3.3333333e+07, -9.9990000e+03,
       -1.2800000e+02,  1.2800000e+02])
>>> y = np.array([complex(np.inf, np.nan), np.nan, complex(np.nan, np.inf)])
array([  1.79769313e+308,  -1.79769313e+308,   0.00000000e+000, # may vary
     -1.28000000e+002,   1.28000000e+002])
>>> np.nan_to_num(y)
```

(continues on next page)

(continued from previous page)

```
array([  1.79769313e+308 +0.00000000e+000j, # may vary
         0.00000000e+000 +0.00000000e+000j,
         0.00000000e+000 +1.79769313e+308j])
>>> np.nan_to_num(y, nan=111111, posinf=222222)
array([222222.+111111.j, 111111.     +0.j, 111111.+222222.j])
```

numpy.**real_if_close**(*a*, *tol=100*)

If input is complex with all imaginary parts close to zero, return real parts.

"Close to zero" is defined as *tol* * (machine epsilon of the type for *a*).

**Parameters**

**a**

[array_like] Input array.

**tol**

[float] Tolerance in machine epsilons for the complex part of the elements in the array.

**Returns**

**out**

[ndarray] If *a* is real, the type of *a* is used for the output. If *a* has complex elements, the returned type is float.

**See also:**

*real*, *imag*, *angle*

## Notes

Machine epsilon varies from machine to machine and between data types but Python floats on most platforms have a machine epsilon equal to 2.2204460492503131e-16. You can use 'np.finfo(float).eps' to print out the machine epsilon for floats.

## Examples

```
>>> np.finfo(float).eps
2.2204460492503131e-16 # may vary
```

```
>>> np.real_if_close([2.1 + 4e-14j, 5.2 + 3e-15j], tol=1000)
array([2.1, 5.2])
>>> np.real_if_close([2.1 + 4e-13j, 5.2 + 3e-15j], tol=1000)
array([2.1+4.e-13j, 5.2 + 3e-15j])
```

numpy.**interp**(*x*, *xp*, *fp*, *left=None*, *right=None*, *period=None*)

One-dimensional linear interpolation.

Returns the one-dimensional piecewise linear interpolant to a function with given discrete data points (*xp*, *fp*), evaluated at *x*.

**Parameters**

**x**

[array_like] The x-coordinates at which to evaluate the interpolated values.

**xp**

[1-D sequence of floats] The x-coordinates of the data points, must be increasing if argument *period* is not specified. Otherwise, *xp* is internally sorted after normalizing the periodic boundaries with `xp = xp % period`.

**fp**

[1-D sequence of float or complex] The y-coordinates of the data points, same length as *xp*.

**left**

[optional float or complex corresponding to fp] Value to return for *x < xp[0]*, default is *fp[0]*.

**right**

[optional float or complex corresponding to fp] Value to return for *x > xp[-1]*, default is *fp[-1]*.

**period**

[None or float, optional] A period for the x-coordinates. This parameter allows the proper interpolation of angular x-coordinates. Parameters *left* and *right* are ignored if *period* is specified.

New in version 1.10.0.

**Returns**

**y**

[float or complex (corresponding to fp) or ndarray] The interpolated values, same shape as *x*.

**Raises**

**ValueError**

If *xp* and *fp* have different length If *xp* or *fp* are not 1-D sequences If *period == 0*

## Notes

The x-coordinate sequence is expected to be increasing, but this is not explicitly enforced. However, if the sequence *xp* is non-increasing, interpolation results are meaningless.

Note that, since NaN is unsortable, *xp* also cannot contain NaNs.

A simple check for *xp* being strictly increasing is:

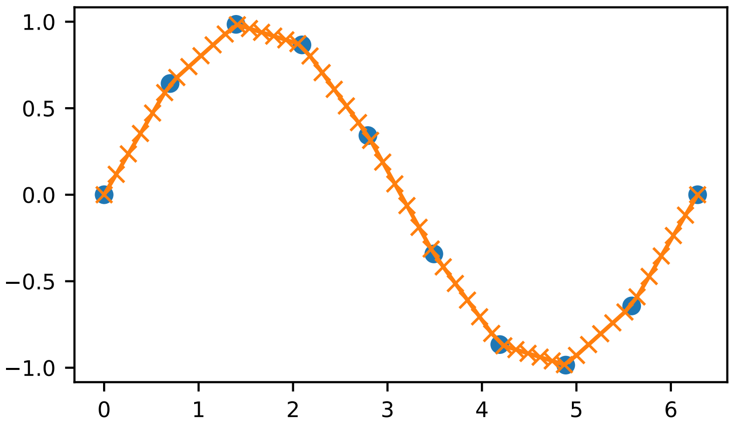
```
np.all(np.diff(xp) > 0)
```

### Examples

```
>>> xp = [1, 2, 3]
>>> fp = [3, 2, 0]
>>> np.interp(2.5, xp, fp)
1.0
>>> np.interp([0, 1, 1.5, 2.72, 3.14], xp, fp)
array([3.  , 3.  , 2.5 , 0.56, 0.  ])
>>> UNDEF = -99.0
>>> np.interp(3.14, xp, fp, right=UNDEF)
-99.0
```

Plot an interpolant to the sine function:

```
>>> x = np.linspace(0, 2*np.pi, 10)
>>> y = np.sin(x)
>>> xvals = np.linspace(0, 2*np.pi, 50)
>>> yinterp = np.interp(xvals, x, y)
>>> import matplotlib.pyplot as plt
>>> plt.plot(x, y, 'o')
[<matplotlib.lines.Line2D object at 0x...>]
>>> plt.plot(xvals, yinterp, '-x')
[<matplotlib.lines.Line2D object at 0x...>]
>>> plt.show()
```



Interpolation with periodic x-coordinates:

```
>>> x = [-180, -170, -185, 185, -10, -5, 0, 365]
>>> xp = [190, -190, 350, -350]
>>> fp = [5, 10, 3, 4]
>>> np.interp(x, xp, fp, period=360)
array([7.5 , 5.  , 8.75, 6.25, 3.  , 3.25, 3.5 , 3.75])
```

Complex interpolation:

```
>>> x = [1.5, 4.0]
>>> xp = [2,3,5]
>>> fp = [1.0j, 0, 2+3j]
>>> np.interp(x, xp, fp)
array([0.+1.j , 1.+1.5j])
```

## 4.20 Matrix library (`numpy.matlib`)

This module contains all functions in the `numpy` namespace, with the following replacement functions that return `matrices` instead of `ndarrays`.

Functions that are also in the numpy namespace and return matrices

| | |
|---|---|
| `mat`(data[, dtype]) | Interpret the input as a matrix. |
| `matrix`(data[, dtype, copy]) | **Note:** It is no longer recommended to use this class, even for linear |
| `asmatrix`(data[, dtype]) | Interpret the input as a matrix. |
| `bmat`(obj[, ldict, gdict]) | Build a matrix object from a string, nested sequence, or array. |

Replacement functions in `matlib`

| | |
|---|---|
| `empty`(shape[, dtype, order]) | Return a new matrix of given shape and type, without initializing entries. |
| `zeros`(shape[, dtype, order]) | Return a matrix of given shape and type, filled with zeros. |
| `ones`(shape[, dtype, order]) | Matrix of ones. |
| `eye`(n[, M, k, dtype, order]) | Return a matrix with ones on the diagonal and zeros elsewhere. |
| `identity`(n[, dtype]) | Returns the square identity matrix of given size. |
| `repmat`(a, m, n) | Repeat a 0-D to 2-D array or matrix MxN times. |
| `rand`(*args) | Return a matrix of random values with given shape. |
| `randn`(*args) | Return a random matrix with data from the “standard normal” distribution. |

`numpy.matlib.`**`empty`**(*shape*, *dtype=None*, *order=’C’*)

Return a new matrix of given shape and type, without initializing entries.

**Parameters**

**shape**

[int or tuple of int] Shape of the empty matrix.

**dtype**

[data-type, optional] Desired output data-type.

**order**

[{'C', 'F'}, optional] Whether to store multi-dimensional data in row-major (C-style) or column-major (Fortran-style) order in memory.

**See also:**

`empty_like`, *`zeros`*

### Notes

*`empty`*, unlike *`zeros`*, does not set the matrix values to zero, and may therefore be marginally faster. On the other hand, it requires the user to manually set all the values in the array, and should be used with caution.

### Examples

```
>>> import numpy.matlib
>>> np.matlib.empty((2, 2))    # filled with random data
matrix([[  6.76425276e-320,   9.79033856e-307], # random
        [  7.39337286e-309,   3.22135945e-309]])
>>> np.matlib.empty((2, 2), dtype=int)
matrix([[ 6600475,        0], # random
        [ 6586976, 22740995]])
```

`numpy.matlib.`**`zeros`**(*shape*, *dtype=None*, *order='C'*)

Return a matrix of given shape and type, filled with zeros.

**Parameters**

**shape**

[int or sequence of ints] Shape of the matrix

**dtype**

[data-type, optional] The desired data-type for the matrix, default is float.

**order**

[{'C', 'F'}, optional] Whether to store the result in C- or Fortran-contiguous order, default is 'C'.

**Returns**

**out**

[matrix] Zero matrix of given shape, dtype, and order.

**See also:**

***`numpy.zeros`***

Equivalent array function.

**`matlib.ones`**

Return a matrix of ones.

### Notes

If `shape` has length one i.e. `(N,)`, or is a scalar `N`, *out* becomes a single row matrix of shape `(1,N)`.

### Examples

```
>>> import numpy.matlib
>>> np.matlib.zeros((2, 3))
matrix([[0.,  0.,  0.],
        [0.,  0.,  0.]])
```

```
>>> np.matlib.zeros(2)
matrix([[0.,  0.]])
```

`numpy.matlib.`**`ones`**`(`*shape*, *dtype=None*, *order='C'*`)`

Matrix of ones.

Return a matrix of given shape and type, filled with ones.

**Parameters**

**shape**

[{sequence of ints, int}] Shape of the matrix

**dtype**

[data-type, optional] The desired data-type for the matrix, default is np.float64.

**order**

[{'C', 'F'}, optional] Whether to store matrix in C- or Fortran-contiguous order, default is 'C'.

**Returns**

**out**

[matrix] Matrix of ones of given shape, dtype, and order.

**See also:**

*`ones`*

Array of ones.

**`matlib.zeros`**

Zero matrix.

### Notes

If `shape` has length one i.e. `(N,)`, or is a scalar `N`, *out* becomes a single row matrix of shape `(1,N)`.

### Examples

```
>>> np.matlib.ones((2,3))
matrix([[1.,  1.,  1.],
        [1.,  1.,  1.]])
```

```
>>> np.matlib.ones(2)
matrix([[1.,  1.]])
```

`numpy.matlib.`**`eye`**`(`*n*, *M=None*, *k=0*, *dtype=<class 'float'>*, *order='C'*`)`

Return a matrix with ones on the diagonal and zeros elsewhere.

**Parameters**

**n**

[int] Number of rows in the output.

**M**

[int, optional] Number of columns in the output, defaults to *n*.

**k**

[int, optional] Index of the diagonal: 0 refers to the main diagonal, a positive value refers to an upper diagonal, and a negative value to a lower diagonal.

**dtype**

[dtype, optional] Data-type of the returned matrix.

**order**

[{'C', 'F'}, optional] Whether the output should be stored in row-major (C-style) or column-major (Fortran-style) order in memory.

New in version 1.14.0.

**Returns**

**I**

[matrix] A *n* x *M* matrix where all elements are equal to zero, except for the *k*-th diagonal, whose values are equal to one.

**See also:**

[`numpy.eye`]

Equivalent array function.

[`identity`]

Square identity matrix.

## Examples

```
>>> import numpy.matlib
>>> np.matlib.eye(3, k=1, dtype=float)
matrix([[0.,  1.,  0.],
        [0.,  0.,  1.],
        [0.,  0.,  0.]])
```

`numpy.matlib.`**`identity`**(*n*, *dtype=None*)

Returns the square identity matrix of given size.

**Parameters**

**n**

[int] Size of the returned identity matrix.

**dtype**

[data-type, optional] Data-type of the output. Defaults to `float`.

**Returns**

**out**

[matrix] *n* x *n* matrix with its main diagonal set to one, and all other elements zero.

**See also:**

***`numpy.identity`***

Equivalent array function.

**`matlib.eye`**

More general matrix identity function.

## Examples

```
>>> import numpy.matlib
>>> np.matlib.identity(3, dtype=int)
matrix([[1, 0, 0],
        [0, 1, 0],
        [0, 0, 1]])
```

`numpy.matlib.`**`repmat`**(*a*, *m*, *n*)

Repeat a 0-D to 2-D array or matrix MxN times.

**Parameters**

**a**

[array_like] The array or matrix to be repeated.

**m, n**

[int] The number of times *a* is repeated along the first and second axes.

**Returns**

**out**

[ndarray] The result of repeating *a*.

### Examples

```
>>> import numpy.matlib
>>> a0 = np.array(1)
>>> np.matlib.repmat(a0, 2, 3)
array([[1, 1, 1],
       [1, 1, 1]])
```

```
>>> a1 = np.arange(4)
>>> np.matlib.repmat(a1, 2, 2)
array([[0, 1, 2, 3, 0, 1, 2, 3],
       [0, 1, 2, 3, 0, 1, 2, 3]])
```

```
>>> a2 = np.asmatrix(np.arange(6).reshape(2, 3))
>>> np.matlib.repmat(a2, 2, 3)
matrix([[0, 1, 2, 0, 1, 2, 0, 1, 2],
        [3, 4, 5, 3, 4, 5, 3, 4, 5],
        [0, 1, 2, 0, 1, 2, 0, 1, 2],
        [3, 4, 5, 3, 4, 5, 3, 4, 5]])
```

`numpy.matlib.`**`rand`**`(`**args*`)`

Return a matrix of random values with given shape.

Create a matrix of the given shape and propagate it with random samples from a uniform distribution over `[0, 1)`.

**Parameters**

***args**

[Arguments] Shape of the output. If given as N integers, each integer specifies the size of one dimension. If given as a tuple, this tuple gives the complete shape.

**Returns**

**out**

[ndarray] The matrix of random values with shape given by **args*.

**See also:**

*`randn`*, *`numpy.random.RandomState.rand`*

## Examples

```
>>> np.random.seed(123)
>>> import numpy.matlib
>>> np.matlib.rand(2, 3)
matrix([[0.69646919, 0.28613933, 0.22685145],
        [0.55131477, 0.71946897, 0.42310646]])
>>> np.matlib.rand((2, 3))
matrix([[0.9807642 , 0.68482974, 0.4809319 ],
        [0.39211752, 0.34317802, 0.72904971]])
```

If the first argument is a tuple, other arguments are ignored:

```
>>> np.matlib.rand((2, 3), 4)
matrix([[0.43857224, 0.0596779 , 0.39804426],
        [0.73799541, 0.18249173, 0.17545176]])
```

`numpy.matlib.`**`randn`**`(`**args*`)`

Return a random matrix with data from the "standard normal" distribution.

*`randn`* generates a matrix filled with random floats sampled from a univariate "normal" (Gaussian) distribution of mean 0 and variance 1.

**Parameters**

***args**

[Arguments] Shape of the output. If given as N integers, each integer specifies the size of one dimension. If given as a tuple, this tuple gives the complete shape.

**Returns**

**Z**

[matrix of floats] A matrix of floating-point samples drawn from the standard normal distribution.

**See also:**

*`rand`*, *`numpy.random.RandomState.randn`*

## Notes

For random samples from $N(\mu, \sigma^2)$, use:

`sigma * np.matlib.randn(...) + mu`

### Examples

```
>>> np.random.seed(123)
>>> import numpy.matlib
>>> np.matlib.randn(1)
matrix([[-1.0856306]])
>>> np.matlib.randn(1, 2, 3)
matrix([[ 0.99734545,  0.2829785 , -1.50629471],
        [-0.57860025,  1.65143654, -2.42667924]])
```

Two-by-four matrix of samples from $N(3, 6.25)$:

```
>>> 2.5 * np.matlib.randn((2, 4)) + 3
matrix([[1.92771843, 6.16484065, 0.83314899, 1.30278462],
        [2.76322758, 6.72847407, 1.40274501, 1.8900451 ]])
```

# 4.21 Miscellaneous routines

## 4.21.1 Performance tuning

| | |
|---|---|
| *setbufsize*(size) | Set the size of the buffer used in ufuncs. |
| *getbufsize*() | Return the size of the buffer used in ufuncs. |

numpy.**getbufsize**()

Return the size of the buffer used in ufuncs.

**Returns**

**getbufsize**

[int] Size of ufunc buffer in bytes.

## 4.21.2 Memory ranges

| | |
|---|---|
| *shares_memory*(a, b[, max_work]) | Determine if two arrays share memory. |
| *may_share_memory*(a, b[, max_work]) | Determine if two arrays might share memory |
| *byte_bounds*(a) | Returns pointers to the end-points of an array. |

numpy.**shares_memory**(*a*, *b*, *max_work=None*)

Determine if two arrays share memory.

> **Warning:** This function can be exponentially slow for some inputs, unless *max_work* is set to a finite number or `MAY_SHARE_BOUNDS`. If in doubt, use *numpy.may_share_memory* instead.

**Parameters**

**a, b**

[ndarray] Input arrays

**max_work**

[int, optional] Effort to spend on solving the overlap problem (maximum number of candidate solutions to consider). The following special values are recognized:

**max_work=MAY_SHARE_EXACT (default)**

The problem is solved exactly. In this case, the function returns True only if there is an element shared between the arrays. Finding the exact solution may take extremely long in some cases.

**max_work=MAY_SHARE_BOUNDS**

Only the memory bounds of a and b are checked.

**Returns**

**out**

[bool]

**Raises**

**numpy.TooHardError**

Exceeded max_work.

**See also:**

*may_share_memory*

### Examples

```
>>> x = np.array([1, 2, 3, 4])
>>> np.shares_memory(x, np.array([5, 6, 7]))
False
>>> np.shares_memory(x[::2], x)
True
>>> np.shares_memory(x[::2], x[1::2])
False
```

Checking whether two arrays share memory is NP-complete, and runtime may increase exponentially in the number of dimensions. Hence, *max_work* should generally be set to a finite number, as it is possible to construct examples that take extremely long to run:

```
>>> from numpy.lib.stride_tricks import as_strided
>>> x = np.zeros([192163377], dtype=np.int8)
>>> x1 = as_strided(x, strides=(36674, 61119, 85569), shape=(1049, 1049, 1049))
>>> x2 = as_strided(x[64023025:], strides=(12223, 12224, 1), shape=(1049, 1049,␣
↪1))
>>> np.shares_memory(x1, x2, max_work=1000)
Traceback (most recent call last):
...
numpy.TooHardError: Exceeded max_work
```

Running `np.shares_memory(x1, x2)` without *max_work* set takes around 1 minute for this case. It is possible to find problems that take still significantly longer.

`numpy.`**`may_share_memory`**`(`*a*, *b*, *max_work=None*`)`

Determine if two arrays might share memory

A return of True does not necessarily mean that the two arrays share any element. It just means that they *might*.

Only the memory bounds of a and b are checked by default.

**Parameters**

**a, b**

[ndarray] Input arrays

**max_work**

[int, optional] Effort to spend on solving the overlap problem. See *`shares_memory`* for details. Default for `may_share_memory` is to do a bounds check.

**Returns**

**out**

[bool]

**See also:**

*`shares_memory`*

## Examples

```
>>> np.may_share_memory(np.array([1,2]), np.array([5,8,9]))
False
>>> x = np.zeros([3, 4])
>>> np.may_share_memory(x[:,0], x[:,1])
True
```

`numpy.`**`byte_bounds`**`(`*a*`)`

Returns pointers to the end-points of an array.

**Parameters**

**a**

[ndarray] Input array. It must conform to the Python-side of the array interface.

**Returns**

**(low, high)**

[tuple of 2 integers] The first integer is the first byte of the array, the second integer is just past the last byte of the array. If *a* is not contiguous it will not use every byte between the (*low*, *high*) values.

### Examples

```
>>> I = np.eye(2, dtype='f'); I.dtype
dtype('float32')
>>> low, high = np.byte_bounds(I)
>>> high - low == I.size*I.itemsize
True
>>> I = np.eye(2); I.dtype
dtype('float64')
>>> low, high = np.byte_bounds(I)
>>> high - low == I.size*I.itemsize
True
```

## 4.21.3 Array mixins

| | |
|---|---|
| *lib.mixins.NDArrayOperatorsMixin* | Mixin defining all operator special methods using __array_ufunc__. |

**class** numpy.lib.mixins.**NDArrayOperatorsMixin**

Mixin defining all operator special methods using __array_ufunc__.

This class implements the special methods for almost all of Python's builtin operators defined in the `operator` module, including comparisons (`==`, `>`, etc.) and arithmetic (`+`, `*`, `-`, etc.), by deferring to the `__array_ufunc__` method, which subclasses must implement.

It is useful for writing classes that do not inherit from *numpy.ndarray*, but that should support arithmetic and numpy universal functions like arrays as described in A Mechanism for Overriding Ufuncs.

As an trivial example, consider this implementation of an `ArrayLike` class that simply wraps a NumPy array and ensures that the result of any arithmetic operation is also an `ArrayLike` object:

```
class ArrayLike(np.lib.mixins.NDArrayOperatorsMixin):
    def __init__(self, value):
        self.value = np.asarray(value)

    # One might also consider adding the built-in list type to this
    # list, to support operations like np.add(array_like, list)
    _HANDLED_TYPES = (np.ndarray, numbers.Number)

    def __array_ufunc__(self, ufunc, method, *inputs, **kwargs):
        out = kwargs.get('out', ())
        for x in inputs + out:
            # Only support operations with instances of _HANDLED_TYPES.
            # Use ArrayLike instead of type(self) for isinstance to
            # allow subclasses that don't override __array_ufunc__ to
            # handle ArrayLike objects.
            if not isinstance(x, self._HANDLED_TYPES + (ArrayLike,)):
                return NotImplemented

        # Defer to the implementation of the ufunc on unwrapped values.
        inputs = tuple(x.value if isinstance(x, ArrayLike) else x
                       for x in inputs)
        if out:
            kwargs['out'] = tuple(
                x.value if isinstance(x, ArrayLike) else x
```

(continues on next page)

(continued from previous page)

```
                for x in out)
        result = getattr(ufunc, method)(*inputs, **kwargs)

        if type(result) is tuple:
            # multiple return values
            return tuple(type(self)(x) for x in result)
        elif method == 'at':
            # no return value
            return None
        else:
            # one return value
            return type(self)(result)

    def __repr__(self):
        return '%s(%r)' % (type(self).__name__, self.value)
```

In interactions between `ArrayLike` objects and numbers or numpy arrays, the result is always another `ArrayLike`:

```
>>> x = ArrayLike([1, 2, 3])
>>> x - 1
ArrayLike(array([0, 1, 2]))
>>> 1 - x
ArrayLike(array([ 0, -1, -2]))
>>> np.arange(3) - x
ArrayLike(array([-1, -1, -1]))
>>> x - np.arange(3)
ArrayLike(array([1, 1, 1]))
```

Note that unlike `numpy.ndarray`, `ArrayLike` does not allow operations with arbitrary, unrecognized types. This ensures that interactions with ArrayLike preserve a well-defined casting hierarchy.

New in version 1.13.

### 4.21.4 NumPy version comparison

| | |
|---|---|
| *lib.NumpyVersion*(vstring) | Parse and compare numpy version strings. |

**class** numpy.lib.**NumpyVersion**(*vstring*)

Parse and compare numpy version strings.

NumPy has the following versioning scheme (numbers given are examples; they can be > 9) in principle):

- Released version: '1.8.0', '1.8.1', etc.
- Alpha: '1.8.0a1', '1.8.0a2', etc.
- Beta: '1.8.0b1', '1.8.0b2', etc.
- Release candidates: '1.8.0rc1', '1.8.0rc2', etc.
- Development versions: '1.8.0.dev-f1234afa' (git commit hash appended)
- **Development versions after a1: '1.8.0a1.dev-f1234afa',**

  '1.8.0b2.dev-f1234afa', '1.8.1rc1.dev-f1234afa', etc.
- Development versions (no git hash available): '1.8.0.dev-Unknown'

Comparing needs to be done against a valid version string or other *NumpyVersion* instance. Note that all development versions of the same (pre-)release compare equal.

New in version 1.9.0.

**Parameters**

**vstring**

[str] NumPy version string (`np.__version__`).

### Examples

```
>>> from numpy.lib import NumpyVersion
>>> if NumpyVersion(np.__version__) < '1.7.0':
...     print('skip')
>>> # skip
```

```
>>> NumpyVersion('1.7')  # raises ValueError, add ".0"
Traceback (most recent call last):
    ...
ValueError: Not a valid numpy version string
```

## 4.21.5 Utility

| | |
|---|---|
| *get_include*() | Return the directory that contains the NumPy *.h header files. |
| *show_config*() | Show libraries in the system on which NumPy was built. |
| *deprecate*(*args, **kwargs) | Issues a DeprecationWarning, adds warning to *old_name*'s docstring, rebinds `old_name.__name__` and returns the new function object. |
| *deprecate_with_doc*(msg) | |

`numpy.`**`get_include`**`()`

Return the directory that contains the NumPy *.h header files.

Extension modules that need to compile against NumPy should use this function to locate the appropriate include directory.

### Notes

When using `distutils`, for example in `setup.py`.

```
import numpy as np
...
Extension('extension_name', ...
        include_dirs=[np.get_include()])
...
```

`numpy.`**`show_config`**`()`

Show libraries in the system on which NumPy was built.

Print information about various resources (libraries, library directories, include directories, etc.) in the system on which NumPy was built.

**See also:**

*get_include*

Returns the directory containing NumPy C header files.

### Notes

Classes specifying the information to be printed are defined in the `numpy.distutils.system_info` module.

Information may include:

- `language`: language used to write the libraries (mostly C or f77)
- `libraries`: names of libraries found in the system
- `library_dirs`: directories containing the libraries
- `include_dirs`: directories containing library header files
- `src_dirs`: directories containing library source files
- `define_macros`: preprocessor macros used by `distutils.setup`

### Examples

```
>>> np.show_config()
blas_opt_info:
    language = c
    define_macros = [('HAVE_CBLAS', None)]
    libraries = ['openblas', 'openblas']
    library_dirs = ['/usr/local/lib']
```

`numpy.`**`deprecate`**`(`**args*, ***kwargs*`)`

Issues a DeprecationWarning, adds warning to *old_name*'s docstring, rebinds `old_name.__name__` and returns the new function object.

This function may also be used as a decorator.

**Parameters**

**func**

[function] The function to be deprecated.

**old_name**

[str, optional] The name of the function to be deprecated. Default is None, in which case the name of *func* is used.

**new_name**

[str, optional] The new name for the function. Default is None, in which case the deprecation message is that *old_name* is deprecated. If given, the deprecation message is that *old_name* is deprecated and *new_name* should be used instead.

**message**

[str, optional] Additional explanation of the deprecation. Displayed in the docstring after the warning.

**Returns**

**old_func**

[function] The deprecated function.

### Examples

Note that `olduint` returns a value after printing Deprecation Warning:

```
>>> olduint = np.deprecate(np.uint)
DeprecationWarning: `uint64` is deprecated! # may vary
>>> olduint(6)
6
```

`numpy.`**`deprecate_with_doc`**`(`*msg*`)`

## 4.21.6 Matlab-like Functions

| | |
|---|---|
| *who*([vardict]) | Print the NumPy arrays in the given dictionary. |
| *disp*(mesg[, device, linefeed]) | Display a message on a device. |

`numpy.`**`who`**`(`*vardict=None*`)`

Print the NumPy arrays in the given dictionary.

If there is no dictionary passed in or *vardict* is None then returns NumPy arrays in the globals() dictionary (all NumPy arrays in the namespace).

**Parameters**

**vardict**

[dict, optional] A dictionary possibly containing ndarrays. Default is globals().

**Returns**

**out**

[None] Returns ‘None’.

## Notes

Prints out the name, shape, bytes and type of all of the ndarrays present in *vardict*.

## Examples

```
>>> a = np.arange(10)
>>> b = np.ones(20)
>>> np.who()
Name                Shape               Bytes               Type
===========================================================
a                   10                  80                  int64
b                   20                  160                 float64
Upper bound on total bytes  =       240
```

```
>>> d = {'x': np.arange(2.0), 'y': np.arange(3.0), 'txt': 'Some str',
... 'idx':5}
>>> np.who(d)
Name                Shape               Bytes               Type
===========================================================
x                   2                   16                  float64
y                   3                   24                  float64
Upper bound on total bytes  =       40
```

`numpy.`**`disp`**(*mesg*, *device=None*, *linefeed=True*)

Display a message on a device.

**Parameters**

**mesg**

[str] Message to display.

**device**

[object] Device to write message. If None, defaults to `sys.stdout` which is very similar to `print`. *device* needs to have `write()` and `flush()` methods.

**linefeed**

[bool, optional] Option whether to print a line feed or not. Defaults to True.

**Raises**

**AttributeError**

If *device* does not have a `write()` or `flush()` method.

### Examples

Besides `sys.stdout`, a file-like object can also be used as it has both required methods:

```
>>> from io import StringIO
>>> buf = StringIO()
>>> np.disp(u'"Display" in a file', device=buf)
>>> buf.getvalue()
'"Display" in a file\n'
```

## 4.22 Padding Arrays

| | |
|---|---|
| *pad*(array, pad_width[, mode]) | Pad an array. |

`numpy.pad`(*array*, *pad_width*, *mode='constant'*, ***kwargs*)

Pad an array.

**Parameters**

**array**

[array_like of rank N] The array to pad.

**pad_width**

[{sequence, array_like, int}] Number of values padded to the edges of each axis. ((before_1, after_1), … (before_N, after_N)) unique pad widths for each axis. ((before, after),) yields same before and after pad for each axis. (pad,) or int is a shortcut for before = after = pad width for all axes.

**mode**

[str or function, optional] One of the following string values or a user supplied function.

**'constant' (default)**

Pads with a constant value.

**'edge'**

Pads with the edge values of array.

**'linear_ramp'**

Pads with the linear ramp between end_value and the array edge value.

**'maximum'**

Pads with the maximum value of all or part of the vector along each axis.

**'mean'**

Pads with the mean value of all or part of the vector along each axis.

**'median'**

Pads with the median value of all or part of the vector along each axis.

**'minimum'**

Pads with the minimum value of all or part of the vector along each axis.

**‘reflect’**

Pads with the reflection of the vector mirrored on the first and last values of the vector along each axis.

**‘symmetric’**

Pads with the reflection of the vector mirrored along the edge of the array.

**‘wrap’**

Pads with the wrap of the vector along the axis. The first values are used to pad the end and the end values are used to pad the beginning.

**‘empty’**

Pads with undefined values.

New in version 1.17.

**<function>**

Padding function, see Notes.

**stat_length**

[sequence or int, optional] Used in ‘maximum’, ‘mean’, ‘median’, and ‘minimum’. Number of values at edge of each axis used to calculate the statistic value.

((before_1, after_1), … (before_N, after_N)) unique statistic lengths for each axis.

((before, after),) yields same before and after statistic lengths for each axis.

(stat_length,) or int is a shortcut for before = after = statistic length for all axes.

Default is `None`, to use the entire axis.

**constant_values**

[sequence or scalar, optional] Used in ‘constant’. The values to set the padded values for each axis.

`((before_1, after_1), ... (before_N, after_N))` unique pad constants for each axis.

`((before, after),)` yields same before and after constants for each axis.

`(constant,)` or `constant` is a shortcut for `before = after = constant` for all axes.

Default is 0.

**end_values**

[sequence or scalar, optional] Used in ‘linear_ramp’. The values used for the ending value of the linear_ramp and that will form the edge of the padded array.

`((before_1, after_1), ... (before_N, after_N))` unique end values for each axis.

`((before, after),)` yields same before and after end values for each axis.

`(constant,)` or `constant` is a shortcut for `before = after = constant` for all axes.

Default is 0.

**reflect_type**

[{'even', 'odd'}, optional] Used in 'reflect', and 'symmetric'. The 'even' style is the default with an unaltered reflection around the edge value. For the 'odd' style, the extended part of the array is created by subtracting the reflected values from two times the edge value.

**Returns**

**pad**

[ndarray] Padded array of rank equal to *`array`* with shape increased according to *pad_width*.

## Notes

New in version 1.7.0.

For an array with rank greater than 1, some of the padding of later axes is calculated from padding of previous axes. This is easiest to think about with a rank 2 array where the corners of the padded array are calculated by using padded values from the first axis.

The padding function, if used, should modify a rank 1 array in-place. It has the following signature:

```
padding_func(vector, iaxis_pad_width, iaxis, kwargs)
```

where

**vector**

[ndarray] A rank 1 array already padded with zeros. Padded values are vector[:iaxis_pad_width[0]] and vector[-iaxis_pad_width[1]:].

**iaxis_pad_width**

[tuple] A 2-tuple of ints, iaxis_pad_width[0] represents the number of values padded at the beginning of vector where iaxis_pad_width[1] represents the number of values padded at the end of vector.

**iaxis**

[int] The axis currently being calculated.

**kwargs**

[dict] Any keyword arguments the function requires.

## Examples

```
>>> a = [1, 2, 3, 4, 5]
>>> np.pad(a, (2, 3), 'constant', constant_values=(4, 6))
array([4, 4, 1, ..., 6, 6, 6])
```

```
>>> np.pad(a, (2, 3), 'edge')
array([1, 1, 1, ..., 5, 5, 5])
```

```
>>> np.pad(a, (2, 3), 'linear_ramp', end_values=(5, -4))
array([ 5,  3,  1,  2,  3,  4,  5,  2, -1, -4])
```

```
>>> np.pad(a, (2,), 'maximum')
array([5, 5, 1, 2, 3, 4, 5, 5, 5])
```

```
>>> np.pad(a, (2,), 'mean')
array([3, 3, 1, 2, 3, 4, 5, 3, 3])
```

```
>>> np.pad(a, (2,), 'median')
array([3, 3, 1, 2, 3, 4, 5, 3, 3])
```

```
>>> a = [[1, 2], [3, 4]]
>>> np.pad(a, ((3, 2), (2, 3)), 'minimum')
array([[1, 1, 1, 2, 1, 1, 1],
       [1, 1, 1, 2, 1, 1, 1],
       [1, 1, 1, 2, 1, 1, 1],
       [1, 1, 1, 2, 1, 1, 1],
       [3, 3, 3, 4, 3, 3, 3],
       [1, 1, 1, 2, 1, 1, 1],
       [1, 1, 1, 2, 1, 1, 1]])
```

```
>>> a = [1, 2, 3, 4, 5]
>>> np.pad(a, (2, 3), 'reflect')
array([3, 2, 1, 2, 3, 4, 5, 4, 3, 2])
```

```
>>> np.pad(a, (2, 3), 'reflect', reflect_type='odd')
array([-1,  0,  1,  2,  3,  4,  5,  6,  7,  8])
```

```
>>> np.pad(a, (2, 3), 'symmetric')
array([2, 1, 1, 2, 3, 4, 5, 5, 4, 3])
```

```
>>> np.pad(a, (2, 3), 'symmetric', reflect_type='odd')
array([0, 1, 1, 2, 3, 4, 5, 5, 6, 7])
```

```
>>> np.pad(a, (2, 3), 'wrap')
array([4, 5, 1, 2, 3, 4, 5, 1, 2, 3])
```

```
>>> def pad_with(vector, pad_width, iaxis, kwargs):
...     pad_value = kwargs.get('padder', 10)
...     vector[:pad_width[0]] = pad_value
...     vector[-pad_width[1]:] = pad_value
>>> a = np.arange(6)
>>> a = a.reshape((2, 3))
>>> np.pad(a, 2, pad_with)
array([[10, 10, 10, 10, 10, 10, 10],
       [10, 10, 10, 10, 10, 10, 10],
       [10, 10,  0,  1,  2, 10, 10],
       [10, 10,  3,  4,  5, 10, 10],
       [10, 10, 10, 10, 10, 10, 10],
       [10, 10, 10, 10, 10, 10, 10]])
>>> np.pad(a, 2, pad_with, padder=100)
array([[100, 100, 100, 100, 100, 100, 100],
       [100, 100, 100, 100, 100, 100, 100],
       [100, 100,   0,   1,   2, 100, 100],
       [100, 100,   3,   4,   5, 100, 100],
```

(continues on next page)

(continued from previous page)

```
       [100, 100, 100, 100, 100, 100, 100],
       [100, 100, 100, 100, 100, 100, 100]])
```

## 4.23 Polynomials

Polynomials in NumPy can be *created*, *manipulated*, and even *fitted* using the *convenience classes* of the `numpy.polynomial` package, introduced in NumPy 1.4.

Prior to NumPy 1.4, `numpy.poly1d` was the class of choice and it is still available in order to maintain backward compatibility. However, the newer Polynomial package is more complete than `numpy.poly1d` and its convenience classes are better behaved in the numpy environment. Therefore `numpy.polynomial` is recommended for new coding.

### 4.23.1 Transition notice

The various routines in the Polynomial package all deal with series whose coefficients go from degree zero upward, which is the *reverse order* of the Poly1d convention. The easy way to remember this is that indexes correspond to degree, i.e., coef[i] is the coefficient of the term of degree i.

#### Using the Convenience Classes

The convenience classes provided by the polynomial package are:

| Name | Provides |
|---|---|
| Polynomial | Power series |
| Chebyshev | Chebyshev series |
| Legendre | Legendre series |
| Laguerre | Laguerre series |
| Hermite | Hermite series |
| HermiteE | HermiteE series |

The series in this context are finite sums of the corresponding polynomial basis functions multiplied by coefficients. For instance, a power series looks like

$$p(x) = 1 + 2x + 3x^2$$

and has coefficients $[1, 2, 3]$. The Chebyshev series with the same coefficients looks like

$$p(x) = 1T_0(x) + 2T_1(x) + 3T_2(x)$$

and more generally

$$p(x) = \sum_{i=0}^{n} c_i T_i(x)$$

where in this case the $T_n$ are the Chebyshev functions of degree $n$, but could just as easily be the basis functions of any of the other classes. The convention for all the classes is that the coefficient $c[i]$ goes with the basis function of degree i.

All of the classes are immutable and have the same methods, and especially they implement the Python numeric operators +, -, *, //, %, divmod, **, ==, and !=. The last two can be a bit problematic due to floating point roundoff errors. We now give a quick demonstration of the various operations using NumPy version 1.7.0.

### Basics

First we need a polynomial class and a polynomial instance to play with. The classes can be imported directly from the polynomial package or from the module of the relevant type. Here we import from the package and use the conventional Polynomial class because of its familiarity:

```
>>> from numpy.polynomial import Polynomial as P
>>> p = P([1,2,3])
>>> p
Polynomial([1., 2., 3.], domain=[-1,  1], window=[-1,  1])
```

Note that there are three parts to the long version of the printout. The first is the coefficients, the second is the domain, and the third is the window:

```
>>> p.coef
array([ 1.,  2.,  3.])
>>> p.domain
array([-1.,  1.])
>>> p.window
array([-1.,  1.])
```

Printing a polynomial yields a shorter form without the domain and window:

```
>>> print(p)
poly([1. 2. 3.])
```

We will deal with the domain and window when we get to fitting, for the moment we ignore them and run through the basic algebraic and arithmetic operations.

Addition and Subtraction:

```
>>> p + p
Polynomial([2., 4., 6.], domain=[-1.,  1.], window=[-1.,  1.])
>>> p - p
Polynomial([0.], domain=[-1.,  1.], window=[-1.,  1.])
```

Multiplication:

```
>>> p * p
Polynomial([ 1.,   4.,  10.,  12.,   9.], domain=[-1.,  1.], window=[-1.,  1.])
```

Powers:

```
>>> p**2
Polynomial([ 1.,   4., 10., 12.,  9.], domain=[-1.,  1.], window=[-1.,  1.])
```

Division:

Floor division, '//', is the division operator for the polynomial classes, polynomials are treated like integers in this regard. For Python versions < 3.x the '/' operator maps to '//', as it does for Python, for later versions the '/' will only work for division by scalars. At some point it will be deprecated:

```
>>> p // P([-1, 1])
Polynomial([5.,  3.], domain=[-1.,  1.], window=[-1.,  1.])
```

Remainder:

```
>>> p % P([-1, 1])
Polynomial([6.], domain=[-1.,  1.], window=[-1.,  1.])
```

Divmod:

```
>>> quo, rem = divmod(p, P([-1, 1]))
>>> quo
Polynomial([5.,  3.], domain=[-1.,  1.], window=[-1.,  1.])
>>> rem
Polynomial([6.], domain=[-1.,  1.], window=[-1.,  1.])
```

Evaluation:

```
>>> x = np.arange(5)
>>> p(x)
array([  1.,   6.,  17.,  34.,  57.])
>>> x = np.arange(6).reshape(3,2)
>>> p(x)
array([[  1.,   6.],
       [ 17.,  34.],
       [ 57.,  86.]])
```

Substitution:

Substitute a polynomial for x and expand the result. Here we substitute p in itself leading to a new polynomial of degree 4 after expansion. If the polynomials are regarded as functions this is composition of functions:

```
>>> p(p)
Polynomial([ 6., 16., 36., 36., 27.], domain=[-1.,  1.], window=[-1.,  1.])
```

Roots:

```
>>> p.roots()
array([-0.33333333-0.47140452j, -0.33333333+0.47140452j])
```

It isn't always convenient to explicitly use Polynomial instances, so tuples, lists, arrays, and scalars are automatically cast in the arithmetic operations:

```
>>> p + [1, 2, 3]
Polynomial([2., 4., 6.], domain=[-1.,  1.], window=[-1.,  1.])
>>> [1, 2, 3] * p
Polynomial([ 1.,  4., 10., 12.,  9.], domain=[-1.,  1.], window=[-1.,  1.])
>>> p / 2
Polynomial([0.5, 1. , 1.5], domain=[-1.,  1.], window=[-1.,  1.])
```

Polynomials that differ in domain, window, or class can't be mixed in arithmetic:

```
>>> from numpy.polynomial import Chebyshev as T
>>> p + P([1], domain=[0,1])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<string>", line 213, in __add__
TypeError: Domains differ
>>> p + P([1], window=[0,1])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<string>", line 215, in __add__
TypeError: Windows differ
>>> p + T([1])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

(continues on next page)

(continued from previous page)

```
  File "<string>", line 211, in __add__
TypeError: Polynomial types differ
```

But different types can be used for substitution. In fact, this is how conversion of Polynomial classes among themselves is done for type, domain, and window casting:

```
>>> p(T([0, 1]))
Chebyshev([2.5, 2. , 1.5], domain=[-1.,  1.], window=[-1.,  1.])
```

Which gives the polynomial *p* in Chebyshev form. This works because $T_1(x) = x$ and substituting $x$ for $x$ doesn't change the original polynomial. However, all the multiplications and divisions will be done using Chebyshev series, hence the type of the result.

It is intended that all polynomial instances are immutable, therefore augmented operations (+=, −=, etc.) and any other functionality that would violate the immutablity of a polynomial instance are intentionally unimplemented.

### Calculus

Polynomial instances can be integrated and differentiated.:

```
>>> from numpy.polynomial import Polynomial as P
>>> p = P([2, 6])
>>> p.integ()
Polynomial([0., 2., 3.], domain=[-1.,  1.], window=[-1.,  1.])
>>> p.integ(2)
Polynomial([0., 0., 1., 1.], domain=[-1.,  1.], window=[-1.,  1.])
```

The first example integrates *p* once, the second example integrates it twice. By default, the lower bound of the integration and the integration constant are 0, but both can be specified.:

```
>>> p.integ(lbnd=-1)
Polynomial([-1.,  2.,  3.], domain=[-1.,  1.], window=[-1.,  1.])
>>> p.integ(lbnd=-1, k=1)
Polynomial([0., 2., 3.], domain=[-1.,  1.], window=[-1.,  1.])
```

In the first case the lower bound of the integration is set to -1 and the integration constant is 0. In the second the constant of integration is set to 1 as well. Differentiation is simpler since the only option is the number of times the polynomial is differentiated:

```
>>> p = P([1, 2, 3])
>>> p.deriv(1)
Polynomial([2., 6.], domain=[-1.,  1.], window=[-1.,  1.])
>>> p.deriv(2)
Polynomial([6.], domain=[-1.,  1.], window=[-1.,  1.])
```

### Other Polynomial Constructors

Constructing polynomials by specifying coefficients is just one way of obtaining a polynomial instance, they may also be created by specifying their roots, by conversion from other polynomial types, and by least squares fits. Fitting is discussed in its own section, the other methods are demonstrated below:

```
>>> from numpy.polynomial import Polynomial as P
>>> from numpy.polynomial import Chebyshev as T
>>> p = P.fromroots([1, 2, 3])
>>> p
Polynomial([-6., 11., -6.,  1.], domain=[-1.,  1.], window=[-1.,  1.])
```

(continues on next page)

(continued from previous page)

```
>>> p.convert(kind=T)
Chebyshev([-9.  , 11.75, -3.  ,  0.25], domain=[-1.,  1.], window=[-1.,  1.])
```

The convert method can also convert domain and window:

```
>>> p.convert(kind=T, domain=[0, 1])
Chebyshev([-2.4375 ,  2.96875, -0.5625 ,  0.03125], domain=[0.,  1.], window=[-1.,  1.
↪])
>>> p.convert(kind=P, domain=[0, 1])
Polynomial([-1.875,  2.875, -1.125,  0.125], domain=[0.,  1.], window=[-1.,  1.])
```

In numpy versions >= 1.7.0 the *basis* and *cast* class methods are also available. The cast method works like the convert method while the basis method returns the basis polynomial of given degree:

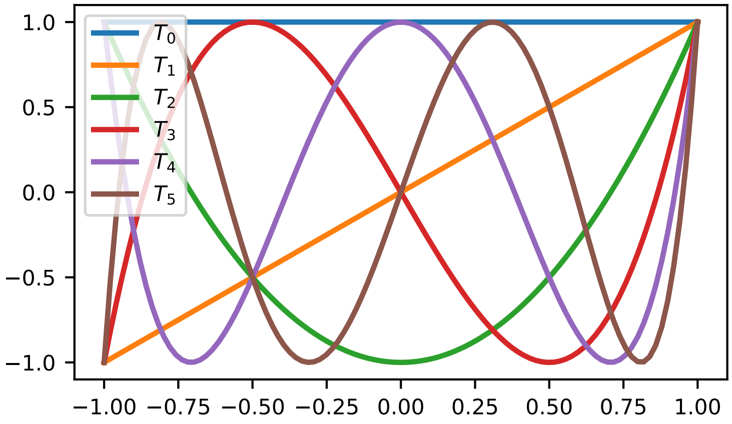
```
>>> P.basis(3)
Polynomial([0., 0., 0., 1.], domain=[-1.,  1.], window=[-1.,  1.])
>>> T.cast(p)
Chebyshev([-9.  , 11.75, -3. ,  0.25], domain=[-1.,  1.], window=[-1.,  1.])
```

Conversions between types can be useful, but it is *not* recommended for routine use. The loss of numerical precision in passing from a Chebyshev series of degree 50 to a Polynomial series of the same degree can make the results of numerical evaluation essentially random.

### Fitting

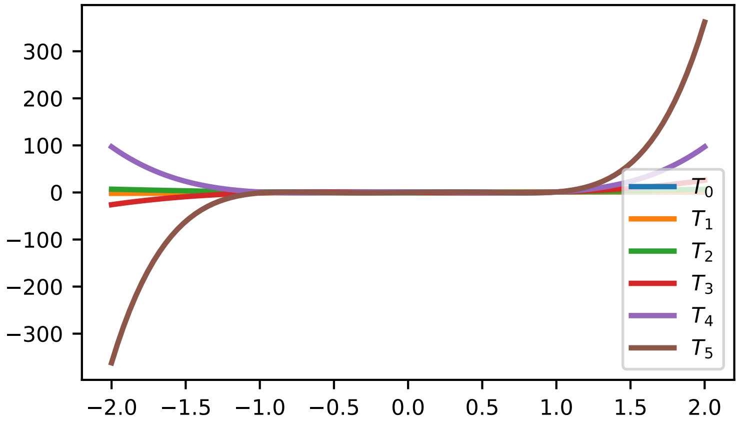
Fitting is the reason that the *domain* and *window* attributes are part of the convenience classes. To illustrate the problem, the values of the Chebyshev polynomials up to degree 5 are plotted below.

```
>>> import matplotlib.pyplot as plt
>>> from numpy.polynomial import Chebyshev as T
>>> x = np.linspace(-1, 1, 100)
>>> for i in range(6): ax = plt.plot(x, T.basis(i)(x), lw=2, label="$T_%d$"%i)
...
>>> plt.legend(loc="upper left")
<matplotlib.legend.Legend object at 0x3b3ee10>
>>> plt.show()
```



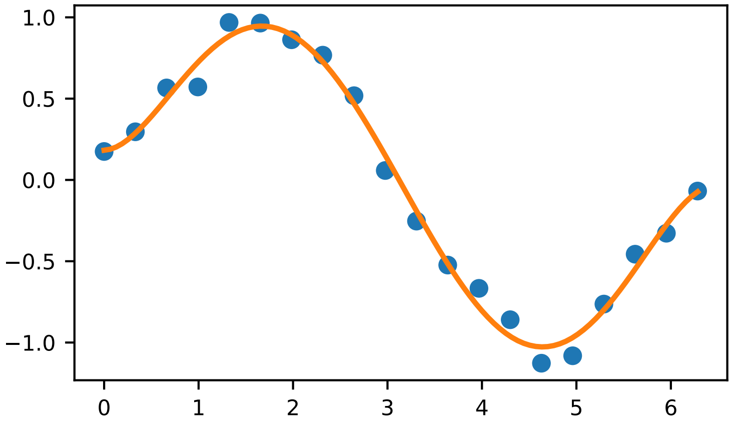
In the range -1 <= *x* <= 1 they are nice, equiripple functions lying between +/- 1. The same plots over the range -2 <= *x* <= 2 look very different:

```
>>> import matplotlib.pyplot as plt
>>> from numpy.polynomial import Chebyshev as T
>>> x = np.linspace(-2, 2, 100)
>>> for i in range(6): ax = plt.plot(x, T.basis(i)(x), lw=2, label="$T_%d$"%i)
...
>>> plt.legend(loc="lower right")
<matplotlib.legend.Legend object at 0x3b3ee10>
>>> plt.show()
```



As can be seen, the "good" parts have shrunk to insignificance. In using Chebyshev polynomials for fitting we want to use the region where *x* is between -1 and 1 and that is what the *window* specifies. However, it is unlikely that the data to be fit has all its data points in that interval, so we use *domain* to specify the interval where the data points lie. When the fit is done, the domain is first mapped to the window by a linear transformation and the usual least squares fit is done using the mapped data points. The window and domain of the fit are part of the returned series and are automatically used when computing values, derivatives, and such. If they aren't specified in the call the fitting routine will use the default window and the smallest domain that holds all the data points. This is illustrated below for a fit to a noisy sine curve.

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> from numpy.polynomial import Chebyshev as T
>>> np.random.seed(11)
>>> x = np.linspace(0, 2*np.pi, 20)
>>> y = np.sin(x) + np.random.normal(scale=.1, size=x.shape)
>>> p = T.fit(x, y, 5)
>>> plt.plot(x, y, 'o')
[<matplotlib.lines.Line2D object at 0x2136c10>]
>>> xx, yy = p.linspace()
>>> plt.plot(xx, yy, lw=2)
[<matplotlib.lines.Line2D object at 0x1cf2890>]
>>> p.domain
array([ 0.        ,  6.28318531])
>>> p.window
array([-1.,  1.])
>>> plt.show()
```



New in version 1.4.0.

## Power Series (`numpy.polynomial.polynomial`)

This module provides a number of objects (mostly functions) useful for dealing with polynomials, including a *`Polynomial`* class that encapsulates the usual arithmetic operations. (General information on how this module represents and works with polynomial objects is in the docstring for its "parent" sub-package, `numpy.polynomial`).

### Classes

| | |
|---|---|
| *`Polynomial`*(coef[, domain, window]) | A power series class. |

**class** `numpy.polynomial.polynomial.`**`Polynomial`**(*coef*, *domain=None*, *window=None*)

A power series class.

The Polynomial class provides the standard Python numerical methods '+', '-', '*', '//', '%', 'divmod', '**', and '()' as well as the attributes and methods listed in the `ABCPolyBase` documentation.

**Parameters**

**coef**

[array_like] Polynomial coefficients in order of increasing degree, i.e., `(1, 2, 3)` give `1 + 2*x + 3*x**2`.

**domain**

[(2,) array_like, optional] Domain to use. The interval `[domain[0], domain[1]]` is mapped to the interval `[window[0], window[1]]` by shifting and scaling. The default value is [-1, 1].

**window**

[(2,) array_like, optional] Window, see `domain` for its use. The default value is [-1, 1].

New in version 1.6.0.

**Attributes**

**basis_name**

### Methods

Table 98 – continued from previous page

| | |
|---|---|
| `has_samecoef`(self, other) | Check if coefficients match. |
| `has_samedomain`(self, other) | Check if domains match. |
| `has_sametype`(self, other) | Check if types match. |
| `has_samewindow`(self, other) | Check if windows match. |
| `identity`([domain, window]) | Identity function. |
| `integ`(self[, m, k, lbnd]) | Integrate. |
| `linspace`(self[, n, domain]) | Return x, y values at equally spaced points in domain. |
| `mapparms`(self) | Return the mapping parameters. |
| `roots`(self) | Return the roots of the series polynomial. |
| `trim`(self[, tol]) | Remove trailing coefficients |
| `truncate`(self, size) | Truncate series to length *size*. |

method

`Polynomial.`**`__call__`**(*self*, *arg*)

Call self as a function.

method

**`classmethod`** `Polynomial.`**`basis`**(*deg*, *domain=None*, *window=None*)

Series basis polynomial of degree *deg*.

Returns the series representing the basis polynomial of degree *deg*.

New in version 1.7.0.

**Parameters**

**deg**

[int] Degree of the basis polynomial for the series. Must be >= 0.

**domain**

[{None, array_like}, optional] If given, the array must be of the form `[beg, end]`, where `beg` and `end` are the endpoints of the domain. If None is given then the class domain is used. The default is None.

**window**

[{None, array_like}, optional] If given, the resulting array must be if the form `[beg, end]`, where `beg` and `end` are the endpoints of the window. If None is given then the class window is used. The default is None.

**Returns**

**new_series**

[series] A series with the coefficient of the *deg* term set to one and all others zero.

method

**`classmethod`** `Polynomial.`**`cast`**(*series*, *domain=None*, *window=None*)

Convert series to series of this class.

The *series* is expected to be an instance of some polynomial series of one of the types supported by by the numpy.polynomial module, but could be some other class that supports the convert method.

New in version 1.7.0.

**Parameters**

**series**

[series] The series instance to be converted.

**domain**

[{None, array_like}, optional] If given, the array must be of the form `[beg, end]`, where `beg` and `end` are the endpoints of the domain. If None is given then the class domain is used. The default is None.

**window**

[{None, array_like}, optional] If given, the resulting array must be if the form `[beg, end]`, where `beg` and `end` are the endpoints of the window. If None is given then the class window is used. The default is None.

**Returns**

**new_series**

[series] A series of the same kind as the calling class and equal to *series* when evaluated.

**See also:**

***`convert`***

similar instance method

method

`Polynomial.`**`convert`**(*self*, *domain=None*, *kind=None*, *window=None*)

Convert series to a different kind and/or domain and/or window.

**Parameters**

**domain**

[array_like, optional] The domain of the converted series. If the value is None, the default domain of *kind* is used.

**kind**

[class, optional] The polynomial series type class to which the current instance should be converted. If kind is None, then the class of the current instance is used.

**window**

[array_like, optional] The window of the converted series. If the value is None, the default window of *kind* is used.

**Returns**

**new_series**

[series] The returned class can be of different type than the current instance and/or have a different domain and/or different window.

### Notes

Conversion between domains and class types can result in numerically ill defined series.

method

`Polynomial.`**`copy`**`(`*self*`)`

Return a copy.

**Returns**

**new_series**

[series] Copy of self.

method

`Polynomial.`**`cutdeg`**`(`*self*, *deg*`)`

Truncate series to the given degree.

Reduce the degree of the series to *deg* by discarding the high order terms. If *deg* is greater than the current degree a copy of the current series is returned. This can be useful in least squares where the coefficients of the high degree terms may be very small.

New in version 1.5.0.

**Parameters**

**deg**

[non-negative int] The series is reduced to degree *deg* by discarding the high order terms. The value of *deg* must be a non-negative integer.

**Returns**

**new_series**

[series] New instance of series with reduced degree.

method

`Polynomial.`**`degree`**`(`*self*`)`

The degree of the series.

New in version 1.5.0.

**Returns**

**degree**

[int] Degree of the series, one less than the number of coefficients.

method

`Polynomial.`**`deriv`**`(`*self*, *m=1*`)`

Differentiate.

Return a series instance of that is the derivative of the current series.

**Parameters**

**m**

[non-negative int] Find the derivative of order *m*.

**Returns**

**new_series**

[series] A new series representing the derivative. The domain is the same as the domain of the differentiated series.

method

**classmethod** `Polynomial.`**`fit`**(*x*, *y*, *deg*, *domain=None*, *rcond=None*, *full=False*, *w=None*, *window=None*)

Least squares fit to data.

Return a series instance that is the least squares fit to the data *y* sampled at *x*. The domain of the returned instance can be specified and this will often result in a superior fit with less chance of ill conditioning.

**Parameters**

**x**

[array_like, shape (M,)] x-coordinates of the M sample points `(x[i], y[i])`.

**y**

[array_like, shape (M,) or (M, K)] y-coordinates of the sample points. Several data sets of sample points sharing the same x-coordinates can be fitted at once by passing in a 2D-array that contains one dataset per column.

**deg**

[int or 1-D array_like] Degree(s) of the fitting polynomials. If *deg* is a single integer all terms up to and including the *deg*'th term are included in the fit. For NumPy versions >= 1.11.0 a list of integers specifying the degrees of the terms to include may be used instead.

**domain**

[{None, [beg, end], []}, optional] Domain to use for the returned series. If `None`, then a minimal domain that covers the points *x* is chosen. If `[]` the class domain is used. The default value was the class domain in NumPy 1.4 and `None` in later versions. The `[]` option was added in numpy 1.5.0.

**rcond**

[float, optional] Relative condition number of the fit. Singular values smaller than this relative to the largest singular value will be ignored. The default value is len(x)*eps, where eps is the relative precision of the float type, about 2e-16 in most cases.

**full**

[bool, optional] Switch determining nature of return value. When it is False (the default) just the coefficients are returned, when True diagnostic information from the singular value decomposition is also returned.

**w**

[array_like, shape (M,), optional] Weights. If not None the contribution of each point `(x[i],y[i])` to the fit is weighted by *w[i]*. Ideally the weights are chosen so that the errors of the products `w[i]*y[i]` all have the same variance. The default value is None.

New in version 1.5.0.

**window**

[{[beg, end]}, optional] Window to use for the returned series. The default value is the default class domain

New in version 1.6.0.

**Returns**

**new_series**

[series] A series that represents the least squares fit to the data and has the domain and window specified in the call. If the coefficients for the unscaled and unshifted basis polynomials are of interest, do `new_series.convert().coef`.

**[resid, rank, sv, rcond]**

[list] These values are only returned if *full* = True

resid – sum of squared residuals of the least squares fit rank – the numerical rank of the scaled Vandermonde matrix sv – singular values of the scaled Vandermonde matrix rcond – value of *rcond*.

For more details, see *linalg.lstsq*.

method

**`classmethod`** `Polynomial.`**`fromroots`**(*roots*, *domain=[]*, *window=None*)

Return series instance that has the specified roots.

Returns a series representing the product `(x - r[0])*(x - r[1])*...*(x - r[n-1])`, where `r` is a list of roots.

**Parameters**

**roots**

[array_like] List of roots.

**domain**

[{[], None, array_like}, optional] Domain for the resulting series. If None the domain is the interval from the smallest root to the largest. If [] the domain is the class domain. The default is [].

**window**

[{None, array_like}, optional] Window for the returned series. If None the class window is used. The default is None.

**Returns**

**new_series**

[series] Series with the specified roots.

method

`Polynomial.`**`has_samecoef`**(*self*, *other*)

Check if coefficients match.

New in version 1.6.0.

**Parameters**

**other**

[class instance] The other class must have the `coef` attribute.

**Returns**

**bool**

[boolean] True if the coefficients are the same, False otherwise.

method

`Polynomial.`**`has_samedomain`**(*self*, *other*)

Check if domains match.

New in version 1.6.0.

**Parameters**

**other**

[class instance] The other class must have the `domain` attribute.

**Returns**

**bool**

[boolean] True if the domains are the same, False otherwise.

method

`Polynomial.`**`has_sametype`**(*self*, *other*)

Check if types match.

New in version 1.7.0.

**Parameters**

**other**

[object] Class instance.

**Returns**

**bool**

[boolean] True if other is same class as self

method

`Polynomial.`**`has_samewindow`**(*self*, *other*)

Check if windows match.

New in version 1.6.0.

**Parameters**

**other**

[class instance] The other class must have the `window` attribute.

**Returns**

**bool**

[boolean] True if the windows are the same, False otherwise.

method

**classmethod** `Polynomial.`**identity**(*domain=None*, *window=None*)

Identity function.

If `p` is the returned series, then `p(x) == x` for all values of x.

**Parameters**

**domain**

[{None, array_like}, optional] If given, the array must be of the form `[beg, end]`, where `beg` and `end` are the endpoints of the domain. If None is given then the class domain is used. The default is None.

**window**

[{None, array_like}, optional] If given, the resulting array must be if the form `[beg, end]`, where `beg` and `end` are the endpoints of the window. If None is given then the class window is used. The default is None.

**Returns**

**new_series**

[series] Series of representing the identity.

method

`Polynomial.`**integ**(*self*, *m=1*, *k=[]*, *lbnd=None*)

Integrate.

Return a series instance that is the definite integral of the current series.

**Parameters**

**m**

[non-negative int] The number of integrations to perform.

**k**

[array_like] Integration constants. The first constant is applied to the first integration, the second to the second, and so on. The list of values must less than or equal to *m* in length and any missing values are set to zero.

**lbnd**

[Scalar] The lower bound of the definite integral.

**Returns**

**new_series**

[series] A new series representing the integral. The domain is the same as the domain of the integrated series.

method

`Polynomial.`**`linspace`**(*self*, *n=100*, *domain=None*)

Return x, y values at equally spaced points in domain.

Returns the x, y values at *n* linearly spaced points across the domain. Here y is the value of the polynomial at the points x. By default the domain is the same as that of the series instance. This method is intended mostly as a plotting aid.

New in version 1.5.0.

**Parameters**

**n**

[int, optional] Number of point pairs to return. The default value is 100.

**domain**

[{None, array_like}, optional] If not None, the specified domain is used instead of that of the calling instance. It should be of the form `[beg,end]`. The default is None which case the class domain is used.

**Returns**

**x, y**

[ndarray] x is equal to linspace(self.domain[0], self.domain[1], n) and y is the series evaluated at element of x.

method

`Polynomial.`**`mapparms`**(*self*)

Return the mapping parameters.

The returned values define a linear map `off + scl*x` that is applied to the input arguments before the series is evaluated. The map depends on the `domain` and `window`; if the current `domain` is equal to the `window` the resulting map is the identity. If the coefficients of the series instance are to be used by themselves outside this class, then the linear function must be substituted for the `x` in the standard representation of the base polynomials.

**Returns**

**off, scl**

[float or complex] The mapping function is defined by `off + scl*x`.

### Notes

If the current domain is the interval `[l1, r1]` and the window is `[l2, r2]`, then the linear mapping function `L` is defined by the equations:

```
L(l1) = l2
L(r1) = r2
```

method

`Polynomial.`**`roots`**(*self*)

Return the roots of the series polynomial.

Compute the roots for the series. Note that the accuracy of the roots decrease the further outside the domain they lie.

**Returns**

**roots**

[ndarray] Array containing the roots of the series.

method

`Polynomial.`**`trim`**`(`*self*, *tol=0*`)`

Remove trailing coefficients

Remove trailing coefficients until a coefficient is reached whose absolute value greater than *tol* or the beginning of the series is reached. If all the coefficients would be removed the series is set to `[0]`. A new series instance is returned with the new coefficients. The current instance remains unchanged.

**Parameters**

**tol**

[non-negative number.] All trailing coefficients less than *tol* will be removed.

**Returns**

**new_series**

[series] Contains the new set of coefficients.

method

`Polynomial.`**`truncate`**`(`*self*, *size*`)`

Truncate series to length *size*.

Reduce the series to length *size* by discarding the high degree terms. The value of *size* must be a positive integer. This can be useful in least squares where the coefficients of the high degree terms may be very small.

**Parameters**

**size**

[positive int] The series is reduced to length *size* by discarding the high degree terms. The value of *size* must be a positive integer.

**Returns**

**new_series**

[series] New instance of series with truncated coefficients.

## Constants

| |
|---|
| *polydomain* |
| *polyzero* |
| *polyone* |
| *polyx* |

`numpy.polynomial.polynomial.`**`polydomain = array([-1, 1])`**

`numpy.polynomial.polynomial.`**`polyzero = array([0])`**

`numpy.polynomial.polynomial.`**`polyone = array([1])`**

`numpy.polynomial.polynomial.`**`polyx = array([0, 1])`**

## Arithmetic

| | |
|---|---|
| *polyadd*(c1, c2) | Add one polynomial to another. |
| *polysub*(c1, c2) | Subtract one polynomial from another. |
| *polymulx*(c) | Multiply a polynomial by x. |
| *polymul*(c1, c2) | Multiply one polynomial by another. |
| *polydiv*(c1, c2) | Divide one polynomial by another. |
| *polypow*(c, pow[, maxpower]) | Raise a polynomial to a power. |
| *polyval*(x, c[, tensor]) | Evaluate a polynomial at points x. |
| *polyval2d*(x, y, c) | Evaluate a 2-D polynomial at points (x, y). |
| *polyval3d*(x, y, z, c) | Evaluate a 3-D polynomial at points (x, y, z). |
| *polygrid2d*(x, y, c) | Evaluate a 2-D polynomial on the Cartesian product of x and y. |
| *polygrid3d*(x, y, z, c) | Evaluate a 3-D polynomial on the Cartesian product of x, y and z. |

`numpy.polynomial.polynomial.`**`polyadd`**(*c1*, *c2*)

Add one polynomial to another.

Returns the sum of two polynomials *c1* + *c2*. The arguments are sequences of coefficients from lowest order term to highest, i.e., [1,2,3] represents the polynomial `1 + 2*x + 3*x**2`.

**Parameters**

**c1, c2**

[array_like] 1-D arrays of polynomial coefficients ordered from low to high.

**Returns**

**out**

[ndarray] The coefficient array representing their sum.

**See also:**

*polysub*, *polymulx*, *polymul*, *polydiv*, *polypow*

### Examples

```
>>> from numpy.polynomial import polynomial as P
>>> c1 = (1,2,3)
>>> c2 = (3,2,1)
>>> sum = P.polyadd(c1,c2); sum
array([4.,  4.,  4.])
>>> P.polyval(2, sum) # 4 + 4(2) + 4(2**2)
28.0
```

`numpy.polynomial.polynomial.`**`polysub`**`(`*c1*, *c2*`)`

Subtract one polynomial from another.

Returns the difference of two polynomials *c1* - *c2*. The arguments are sequences of coefficients from lowest order term to highest, i.e., [1,2,3] represents the polynomial `1 + 2*x + 3*x**2`.

**Parameters**

**c1, c2**

[array_like] 1-D arrays of polynomial coefficients ordered from low to high.

**Returns**

**out**

[ndarray] Of coefficients representing their difference.

**See also:**

*`polyadd`*, *`polymulx`*, *`polymul`*, *`polydiv`*, *`polypow`*

### Examples

```
>>> from numpy.polynomial import polynomial as P
>>> c1 = (1,2,3)
>>> c2 = (3,2,1)
>>> P.polysub(c1,c2)
array([-2.,  0.,  2.])
>>> P.polysub(c2,c1) # -P.polysub(c1,c2)
array([ 2.,  0., -2.])
```

`numpy.polynomial.polynomial.`**`polymulx`**`(`*c*`)`

Multiply a polynomial by x.

Multiply the polynomial *c* by x, where x is the independent variable.

**Parameters**

**c**

[array_like] 1-D array of polynomial coefficients ordered from low to high.

**Returns**

**out**

[ndarray] Array representing the result of the multiplication.

**See also:**

`polyadd`, `polysub`, `polymul`, `polydiv`, `polypow`

### Notes

New in version 1.5.0.

`numpy.polynomial.polynomial.`**`polymul`**`(`*c1*, *c2*`)`

Multiply one polynomial by another.

Returns the product of two polynomials *c1* * *c2*. The arguments are sequences of coefficients, from lowest order term to highest, e.g., [1,2,3] represents the polynomial `1 + 2*x + 3*x**2.`

**Parameters**

**c1, c2**

[array_like] 1-D arrays of coefficients representing a polynomial, relative to the "standard" basis, and ordered from lowest order term to highest.

**Returns**

**out**

[ndarray] Of the coefficients of their product.

**See also:**

`polyadd`, `polysub`, `polymulx`, `polydiv`, `polypow`

### Examples

```
>>> from numpy.polynomial import polynomial as P
>>> c1 = (1,2,3)
>>> c2 = (3,2,1)
>>> P.polymul(c1,c2)
array([  3.,   8.,  14.,   8.,   3.])
```

`numpy.polynomial.polynomial.`**`polydiv`**`(`*c1*, *c2*`)`

Divide one polynomial by another.

Returns the quotient-with-remainder of two polynomials *c1* / *c2*. The arguments are sequences of coefficients, from lowest order term to highest, e.g., [1,2,3] represents `1 + 2*x + 3*x**2`.

**Parameters**

**c1, c2**

[array_like] 1-D arrays of polynomial coefficients ordered from low to high.

**Returns**

**[quo, rem]**

[ndarrays] Of coefficient series representing the quotient and remainder.

**See also:**

*polyadd*, *polysub*, *polymulx*, *polymul*, *polypow*

### Examples

```
>>> from numpy.polynomial import polynomial as P
>>> c1 = (1,2,3)
>>> c2 = (3,2,1)
>>> P.polydiv(c1,c2)
(array([3.]), array([-8., -4.]))
>>> P.polydiv(c2,c1)
(array([ 0.33333333]), array([ 2.66666667,  1.33333333])) # may vary
```

`numpy.polynomial.polynomial.`**`polypow`**(*c*, *pow*, *maxpower=None*)

Raise a polynomial to a power.

Returns the polynomial *c* raised to the power *pow*. The argument *c* is a sequence of coefficients ordered from low to high. i.e., [1,2,3] is the series `1 + 2*x + 3*x**2.`

**Parameters**

**c**

[array_like] 1-D array of array of series coefficients ordered from low to high degree.

**pow**

[integer] Power to which the series will be raised

**maxpower**

[integer, optional] Maximum power allowed. This is mainly to limit growth of the series to unmanageable size. Default is 16

**Returns**

**coef**

[ndarray] Power series of power.

**See also:**

*polyadd*, *polysub*, *polymulx*, *polymul*, *polydiv*

### Examples

```
>>> from numpy.polynomial import polynomial as P
>>> P.polypow([1,2,3], 2)
array([ 1., 4., 10., 12., 9.])
```

`numpy.polynomial.polynomial.`**`polyval`**(*x*, *c*, *tensor=True*)

Evaluate a polynomial at points x.

If *c* is of length *n + 1*, this function returns the value

$$p(x) = c_0 + c_1 * x + ... + c_n * x^n$$

The parameter *x* is converted to an array only if it is a tuple or a list, otherwise it is treated as a scalar. In either case, either *x* or its elements must support multiplication and addition both with themselves and with the elements of *c*.

If *c* is a 1-D array, then *p(x)* will have the same shape as *x*. If *c* is multidimensional, then the shape of the result depends on the value of *tensor*. If *tensor* is true the shape will be c.shape[1:] + x.shape. If *tensor* is false the shape will be c.shape[1:]. Note that scalars have shape (,).

Trailing zeros in the coefficients will be used in the evaluation, so they should be avoided if efficiency is a concern.

**Parameters**

**x**

[array_like, compatible object] If *x* is a list or tuple, it is converted to an ndarray, otherwise it is left unchanged and treated as a scalar. In either case, *x* or its elements must support addition and multiplication with with themselves and with the elements of *c*.

**c**

[array_like] Array of coefficients ordered so that the coefficients for terms of degree n are contained in c[n]. If *c* is multidimensional the remaining indices enumerate multiple polynomials. In the two dimensional case the coefficients may be thought of as stored in the columns of *c*.

**tensor**

[boolean, optional] If True, the shape of the coefficient array is extended with ones on the right, one for each dimension of *x*. Scalars have dimension 0 for this action. The result is that every column of coefficients in *c* is evaluated for every element of *x*. If False, *x* is broadcast over the columns of *c* for the evaluation. This keyword is useful when *c* is multidimensional. The default value is True.

New in version 1.7.0.

**Returns**

**values**

[ndarray, compatible object] The shape of the returned array is described above.

**See also:**

`polyval2d`, `polygrid2d`, `polyval3d`, `polygrid3d`

## Notes

The evaluation uses Horner's method.

### Examples

```
>>> from numpy.polynomial.polynomial import polyval
>>> polyval(1, [1,2,3])
6.0
>>> a = np.arange(4).reshape(2,2)
>>> a
array([[0, 1],
       [2, 3]])
>>> polyval(a, [1,2,3])
array([[ 1.,   6.],
       [17.,  34.]])
>>> coef = np.arange(4).reshape(2,2) # multidimensional coefficients
>>> coef
array([[0, 1],
       [2, 3]])
>>> polyval([1,2], coef, tensor=True)
array([[2.,  4.],
       [4.,  7.]])
>>> polyval([1,2], coef, tensor=False)
array([2.,  7.])
```

`numpy.polynomial.polynomial.`**`polyval2d`**`(`*x*, *y*, *c*`)`

Evaluate a 2-D polynomial at points (x, y).

This function returns the value

$$p(x,y) = \sum_{i,j} c_{i,j} * x^i * y^j$$

The parameters *x* and *y* are converted to arrays only if they are tuples or a lists, otherwise they are treated as a scalars and they must have the same shape after conversion. In either case, either *x* and *y* or their elements must support multiplication and addition both with themselves and with the elements of *c*.

If *c* has fewer than two dimensions, ones are implicitly appended to its shape to make it 2-D. The shape of the result will be c.shape[2:] + x.shape.

**Parameters**

**x, y**

[array_like, compatible objects] The two dimensional series is evaluated at the points *(x, y)*, where *x* and *y* must have the same shape. If *x* or *y* is a list or tuple, it is first converted to an ndarray, otherwise it is left unchanged and, if it isn't an ndarray, it is treated as a scalar.

**c**

[array_like] Array of coefficients ordered so that the coefficient of the term of multi-degree i,j is contained in *c[i,j]*. If *c* has dimension greater than two the remaining indices enumerate multiple sets of coefficients.

**Returns**

**values**

[ndarray, compatible object] The values of the two dimensional polynomial at points formed with pairs of corresponding values from *x* and *y*.

**See also:**

*`polyval`*, *`polygrid2d`*, *`polyval3d`*, *`polygrid3d`*

### Notes

New in version 1.7.0.

`numpy.polynomial.polynomial.`**`polyval3d`**`(`*x*, *y*, *z*, *c*`)`

Evaluate a 3-D polynomial at points (x, y, z).

This function returns the values:

$$p(x,y,z) = \sum_{i,j,k} c_{i,j,k} * x^i * y^j * z^k$$

The parameters *x*, *y*, and *z* are converted to arrays only if they are tuples or a lists, otherwise they are treated as a scalars and they must have the same shape after conversion. In either case, either *x*, *y*, and *z* or their elements must support multiplication and addition both with themselves and with the elements of *c*.

If *c* has fewer than 3 dimensions, ones are implicitly appended to its shape to make it 3-D. The shape of the result will be c.shape[3:] + x.shape.

**Parameters**

**x, y, z**

[array_like, compatible object] The three dimensional series is evaluated at the points *(x, y, z)*, where *x*, *y*, and *z* must have the same shape. If any of *x*, *y*, or *z* is a list or tuple, it is first converted to an ndarray, otherwise it is left unchanged and if it isn't an ndarray it is treated as a scalar.

**c**

[array_like] Array of coefficients ordered so that the coefficient of the term of multi-degree i,j,k is contained in `c[i,j,k]`. If *c* has dimension greater than 3 the remaining indices enumerate multiple sets of coefficients.

**Returns**

**values**

[ndarray, compatible object] The values of the multidimensional polynomial on points formed with triples of corresponding values from *x*, *y*, and *z*.

**See also:**

*`polyval`*, *`polyval2d`*, *`polygrid2d`*, *`polygrid3d`*

### Notes

New in version 1.7.0.

`numpy.polynomial.polynomial.`**`polygrid2d`**`(`*x*, *y*, *c*`)`

Evaluate a 2-D polynomial on the Cartesian product of x and y.

This function returns the values:

$$p(a,b) = \sum_{i,j} c_{i,j} * a^i * b^j$$

where the points *(a, b)* consist of all pairs formed by taking *a* from *x* and *b* from *y*. The resulting points form a grid with *x* in the first dimension and *y* in the second.

The parameters *x* and *y* are converted to arrays only if they are tuples or a lists, otherwise they are treated as a scalars. In either case, either *x* and *y* or their elements must support multiplication and addition both with themselves and with the elements of *c*.

If *c* has fewer than two dimensions, ones are implicitly appended to its shape to make it 2-D. The shape of the result will be c.shape[2:] + x.shape + y.shape.

**Parameters**

**x, y**

[array_like, compatible objects] The two dimensional series is evaluated at the points in the Cartesian product of *x* and *y*. If *x* or *y* is a list or tuple, it is first converted to an ndarray, otherwise it is left unchanged and, if it isn't an ndarray, it is treated as a scalar.

**c**

[array_like] Array of coefficients ordered so that the coefficients for terms of degree i,j are contained in `c[i,j]`. If *c* has dimension greater than two the remaining indices enumerate multiple sets of coefficients.

**Returns**

**values**

[ndarray, compatible object] The values of the two dimensional polynomial at points in the Cartesian product of *x* and *y*.

**See also:**

*polyval*, *polyval2d*, *polyval3d*, *polygrid3d*

## Notes

New in version 1.7.0.

`numpy.polynomial.polynomial.`**`polygrid3d`**`(`*x, y, z, c*`)`

Evaluate a 3-D polynomial on the Cartesian product of x, y and z.

This function returns the values:

$$p(a,b,c) = \sum_{i,j,k} c_{i,j,k} * a^i * b^j * c^k$$

where the points *(a, b, c)* consist of all triples formed by taking *a* from *x*, *b* from *y*, and *c* from *z*. The resulting points form a grid with *x* in the first dimension, *y* in the second, and *z* in the third.

The parameters *x*, *y*, and *z* are converted to arrays only if they are tuples or a lists, otherwise they are treated as a scalars. In either case, either *x*, *y*, and *z* or their elements must support multiplication and addition both with themselves and with the elements of *c*.

If *c* has fewer than three dimensions, ones are implicitly appended to its shape to make it 3-D. The shape of the result will be c.shape[3:] + x.shape + y.shape + z.shape.

**Parameters**

**x, y, z**

[array_like, compatible objects] The three dimensional series is evaluated at the points in the Cartesian product of *x*, *y*, and *z*. If *x*,'y', or *z* is a list or tuple, it is first converted to an ndarray, otherwise it is left unchanged and, if it isn't an ndarray, it is treated as a scalar.

**c**

[array_like] Array of coefficients ordered so that the coefficients for terms of degree i,j are contained in `c[i,j]`. If *c* has dimension greater than two the remaining indices enumerate multiple sets of coefficients.

**Returns**

**values**

[ndarray, compatible object] The values of the two dimensional polynomial at points in the Cartesian product of *x* and *y*.

**See also:**

*polyval*, *polyval2d*, *polygrid2d*, *polyval3d*

### Notes

New in version 1.7.0.

## Calculus

| | |
|---|---|
| *polyder*(c[, m, scl, axis]) | Differentiate a polynomial. |
| *polyint*(c[, m, k, lbnd, scl, axis]) | Integrate a polynomial. |

`numpy.polynomial.polynomial.`**`polyder`**`(`*c*, *m=1*, *scl=1*, *axis=0*`)`

Differentiate a polynomial.

Returns the polynomial coefficients *c* differentiated *m* times along *axis*. At each iteration the result is multiplied by *scl* (the scaling factor is for use in a linear change of variable). The argument *c* is an array of coefficients from low to high degree along each axis, e.g., [1,2,3] represents the polynomial `1 + 2*x + 3*x**2` while [[1,2],[1,2]] represents `1 + 1*x + 2*y + 2*x*y` if axis=0 is `x` and axis=1 is `y`.

**Parameters**

**c**

[array_like] Array of polynomial coefficients. If c is multidimensional the different axis correspond to different variables with the degree in each axis given by the corresponding index.

**m**

[int, optional] Number of derivatives taken, must be non-negative. (Default: 1)

**scl**

[scalar, optional] Each differentiation is multiplied by *scl*. The end result is multiplication by `scl**m`. This is for use in a linear change of variable. (Default: 1)

**axis**

[int, optional] Axis over which the derivative is taken. (Default: 0).

New in version 1.7.0.

**Returns**

**der**

[ndarray] Polynomial coefficients of the derivative.

**See also:**

`polyint`

### Examples

```
>>> from numpy.polynomial import polynomial as P
>>> c = (1,2,3,4) # 1 + 2x + 3x**2 + 4x**3
>>> P.polyder(c) # (d/dx)(c) = 2 + 6x + 12x**2
array([  2.,   6.,  12.])
>>> P.polyder(c,3) # (d**3/dx**3)(c) = 24
array([24.])
>>> P.polyder(c,scl=-1) # (d/d(-x))(c) = -2 - 6x - 12x**2
array([ -2.,  -6., -12.])
>>> P.polyder(c,2,-1) # (d**2/d(-x)**2)(c) = 6 + 24x
array([  6.,  24.])
```

`numpy.polynomial.polynomial.`**`polyint`**(*c*, *m=1*, *k=[]*, *lbnd=0*, *scl=1*, *axis=0*)

Integrate a polynomial.

Returns the polynomial coefficients *c* integrated *m* times from *lbnd* along *axis*. At each iteration the resulting series is **multiplied** by *scl* and an integration constant, *k*, is added. The scaling factor is for use in a linear change of variable. ("Buyer beware": note that, depending on what one is doing, one may want *scl* to be the reciprocal of what one might expect; for more information, see the Notes section below.) The argument *c* is an array of coefficients, from low to high degree along each axis, e.g., [1,2,3] represents the polynomial `1 + 2*x + 3*x**2` while [[1,2],[1,2]] represents `1 + 1*x + 2*y + 2*x*y` if axis=0 is `x` and axis=1 is `y`.

**Parameters**

**c**

[array_like] 1-D array of polynomial coefficients, ordered from low to high.

**m**

[int, optional] Order of integration, must be positive. (Default: 1)

**k**

[{[], list, scalar}, optional] Integration constant(s). The value of the first integral at zero is the first value in the list, the value of the second integral at zero is the second value, etc. If `k == []` (the default), all constants are set to zero. If `m == 1`, a single scalar can be given instead of a list.

**lbnd**

[scalar, optional] The lower bound of the integral. (Default: 0)

**scl**

[scalar, optional] Following each integration the result is *multiplied* by *scl* before the integration constant is added. (Default: 1)

**axis**

[int, optional] Axis over which the integral is taken. (Default: 0).

New in version 1.7.0.

**Returns**

**S**

[ndarray] Coefficient array of the integral.

**Raises**

**ValueError**

If `m < 1`, `len(k) > m`, `np.ndim(lbnd) != 0`, or `np.ndim(scl) != 0`.

**See also:**

*`polyder`*

## Notes

Note that the result of each integration is *multiplied* by *scl*. Why is this important to note? Say one is making a linear change of variable $u = ax + b$ in an integral relative to *x*. Then $dx = du/a$, so one will need to set *scl* equal to $1/a$ - perhaps not what one would have first thought.

## Examples

```
>>> from numpy.polynomial import polynomial as P
>>> c = (1,2,3)
>>> P.polyint(c) # should return array([0, 1, 1, 1])
array([0.,  1.,  1.,  1.])
>>> P.polyint(c,3) # should return array([0, 0, 0, 1/6, 1/12, 1/20])
 array([ 0.        ,  0.        ,  0.        ,  0.16666667,  0.08333333, # may␣
↪vary
        0.05      ])
>>> P.polyint(c,k=3) # should return array([3, 1, 1, 1])
array([3.,  1.,  1.,  1.])
>>> P.polyint(c,lbnd=-2) # should return array([6, 1, 1, 1])
array([6.,  1.,  1.,  1.])
>>> P.polyint(c,scl=-2) # should return array([0, -2, -2, -2])
array([ 0., -2., -2., -2.])
```

## Misc Functions

| | |
|---|---|
| *`polyfromroots`*(roots) | Generate a monic polynomial with given roots. |
| *`polyroots`*(c) | Compute the roots of a polynomial. |
| *`polyvalfromroots`*(x, r[, tensor]) | Evaluate a polynomial specified by its roots at points x. |
| *`polyvander`*(x, deg) | Vandermonde matrix of given degree. |
| *`polyvander2d`*(x, y, deg) | Pseudo-Vandermonde matrix of given degrees. |
| *`polyvander3d`*(x, y, z, deg) | Pseudo-Vandermonde matrix of given degrees. |
| *`polycompanion`*(c) | Return the companion matrix of c. |
| *`polyfit`*(x, y, deg[, rcond, full, w]) | Least-squares fit of a polynomial to data. |
| *`polytrim`*(c[, tol]) | Remove "small" "trailing" coefficients from a polynomial. |
| *`polyline`*(off, scl) | Returns an array representing a linear polynomial. |

`numpy.polynomial.polynomial.`**`polyfromroots`**(*roots*)

Generate a monic polynomial with given roots.

Return the coefficients of the polynomial

$$p(x) = (x - r_0) * (x - r_1) * ... * (x - r_n),$$

where the *r_n* are the roots specified in *roots*. If a zero has multiplicity n, then it must appear in *roots* n times. For instance, if 2 is a root of multiplicity three and 3 is a root of multiplicity 2, then *roots* looks something like [2, 2, 2, 3, 3]. The roots can appear in any order.

If the returned coefficients are *c*, then

$$p(x) = c_0 + c_1 * x + ... + x^n$$

The coefficient of the last term is 1 for monic polynomials in this form.

**Parameters**

**roots**

[array_like] Sequence containing the roots.

**Returns**

**out**

[ndarray] 1-D array of the polynomial's coefficients If all the roots are real, then *out* is also real, otherwise it is complex. (see Examples below).

**See also:**

`chebfromroots`, `legfromroots`, `lagfromroots`, `hermfromroots`, `hermefromroots`

## Notes

The coefficients are determined by multiplying together linear factors of the form *(x - r_i)*, i.e.

$$p(x) = (x - r_0)(x - r_1)...(x - r_n)$$

where `n == len(roots) - 1`; note that this implies that *1* is always returned for $a_n$.

## Examples

```
>>> from numpy.polynomial import polynomial as P
>>> P.polyfromroots((-1,0,1)) # x(x - 1)(x + 1) = x^3 - x
array([ 0., -1.,  0.,  1.])
>>> j = complex(0,1)
>>> P.polyfromroots((-j,j)) # complex returned, though values are real
array([1.+0.j,  0.+0.j,  1.+0.j])
```

`numpy.polynomial.polynomial.`**`polyroots`**`(c)`

Compute the roots of a polynomial.

Return the roots (a.k.a. "zeros") of the polynomial

$$p(x) = \sum_i c[i] * x^i.$$

**Parameters**

**c**

[1-D array_like] 1-D array of polynomial coefficients.

**Returns**

**out**

[ndarray] Array of the roots of the polynomial. If all the roots are real, then *out* is also real, otherwise it is complex.

**See also:**

`chebroots`

## Notes

The root estimates are obtained as the eigenvalues of the companion matrix, Roots far from the origin of the complex plane may have large errors due to the numerical instability of the power series for such values. Roots with multiplicity greater than 1 will also show larger errors as the value of the series near such points is relatively insensitive to errors in the roots. Isolated roots near the origin can be improved by a few iterations of Newton's method.

## Examples

```
>>> import numpy.polynomial.polynomial as poly
>>> poly.polyroots(poly.polyfromroots((-1,0,1)))
array([-1.,  0.,  1.])
>>> poly.polyroots(poly.polyfromroots((-1,0,1))).dtype
dtype('float64')
>>> j = complex(0,1)
>>> poly.polyroots(poly.polyfromroots((-j,0,j)))
array([  0.00000000e+00+0.j,   0.00000000e+00+1.j,   2.77555756e-17-1.j]) # may␣
↪vary
```

`numpy.polynomial.polynomial.`**`polyvalfromroots`**(*x*, *r*, *tensor=True*)

Evaluate a polynomial specified by its roots at points x.

If *r* is of length *N*, this function returns the value

$$p(x) = \prod_{n=1}^{N} (x - r_n)$$

The parameter *x* is converted to an array only if it is a tuple or a list, otherwise it is treated as a scalar. In either case, either *x* or its elements must support multiplication and addition both with themselves and with the elements of *r*.

If *r* is a 1-D array, then *p(x)* will have the same shape as *x*. If *r* is multidimensional, then the shape of the result depends on the value of *tensor*. If *tensor is "True'* the shape will be r.shape[1:] + x.shape; that is, each polynomial is evaluated at every value of *x*. If *tensor* is `False`, the shape will be r.shape[1:]; that is, each polynomial is evaluated only for the corresponding broadcast value of *x*. Note that scalars have shape (,).

New in version 1.12.

**Parameters**

**x**

[array_like, compatible object] If *x* is a list or tuple, it is converted to an ndarray, otherwise it is left unchanged and treated as a scalar. In either case, *x* or its elements must support addition and multiplication with with themselves and with the elements of *r*.

**r**

[array_like] Array of roots. If *r* is multidimensional the first index is the root index, while the remaining indices enumerate multiple polynomials. For instance, in the two dimensional case the roots of each polynomial may be thought of as stored in the columns of *r*.

**tensor**

[boolean, optional] If True, the shape of the roots array is extended with ones on the right, one for each dimension of *x*. Scalars have dimension 0 for this action. The result is that every column of coefficients in *r* is evaluated for every element of *x*. If False, *x* is broadcast over the columns of *r* for the evaluation. This keyword is useful when *r* is multidimensional. The default value is True.

**Returns**

**values**

[ndarray, compatible object] The shape of the returned array is described above.

**See also:**

*polyroots*, *polyfromroots*, *polyval*

## Examples

```
>>> from numpy.polynomial.polynomial import polyvalfromroots
>>> polyvalfromroots(1, [1,2,3])
0.0
>>> a = np.arange(4).reshape(2,2)
>>> a
array([[0, 1],
       [2, 3]])
>>> polyvalfromroots(a, [-1, 0, 1])
array([[-0.,   0.],
       [ 6.,  24.]])
>>> r = np.arange(-2, 2).reshape(2,2) # multidimensional coefficients
>>> r # each column of r defines one polynomial
array([[-2, -1],
       [ 0,  1]])
>>> b = [-2, 1]
>>> polyvalfromroots(b, r, tensor=True)
array([[-0.,  3.],
       [ 3., 0.]])
>>> polyvalfromroots(b, r, tensor=False)
array([-0.,  0.])
```

`numpy.polynomial.polynomial.`**`polyvander`**(*x*, *deg*)

Vandermonde matrix of given degree.

Returns the Vandermonde matrix of degree *deg* and sample points *x*. The Vandermonde matrix is defined by

$$V[..., i] = x^i,$$

where *0 <= i <= deg*. The leading indices of *V* index the elements of *x* and the last index is the power of *x*.

If *c* is a 1-D array of coefficients of length *n + 1* and *V* is the matrix `V = polyvander(x, n)`, then `np.dot(V, c)` and `polyval(x, c)` are the same up to roundoff. This equivalence is useful both for least squares fitting and for the evaluation of a large number of polynomials of the same degree and sample points.

**Parameters**

**x**

[array_like] Array of points. The dtype is converted to float64 or complex128 depending on whether any of the elements are complex. If *x* is scalar it is converted to a 1-D array.

**deg**

[int] Degree of the resulting matrix.

**Returns**

**vander**

[ndarray.] The Vandermonde matrix. The shape of the returned matrix is `x.shape + (deg + 1,)`, where the last index is the power of *x*. The dtype will be the same as the converted *x*.

**See also:**

*`polyvander2d`*, *`polyvander3d`*

`numpy.polynomial.polynomial.`**`polyvander2d`**`(`*x*, *y*, *deg*`)`

Pseudo-Vandermonde matrix of given degrees.

Returns the pseudo-Vandermonde matrix of degrees *deg* and sample points *(x, y)*. The pseudo-Vandermonde matrix is defined by

$$V[..., (deg[1] + 1) * i + j] = x^i * y^j,$$

where *0 <= i <= deg[0]* and *0 <= j <= deg[1]*. The leading indices of *V* index the points *(x, y)* and the last index encodes the powers of *x* and *y*.

If `V = polyvander2d(x, y, [xdeg, ydeg])`, then the columns of *V* correspond to the elements of a 2-D coefficient array *c* of shape (xdeg + 1, ydeg + 1) in the order

$$c_{00}, c_{01}, c_{02}..., c_{10}, c_{11}, c_{12}...$$

and `np.dot(V, c.flat)` and `polyval2d(x, y, c)` will be the same up to roundoff. This equivalence is useful both for least squares fitting and for the evaluation of a large number of 2-D polynomials of the same degrees and sample points.

**Parameters**

**x, y**

[array_like] Arrays of point coordinates, all of the same shape. The dtypes will be converted to either float64 or complex128 depending on whether any of the elements are complex. Scalars are converted to 1-D arrays.

**deg**

[list of ints] List of maximum degrees of the form [x_deg, y_deg].

**Returns**

**vander2d**

[ndarray] The shape of the returned matrix is `x.shape + (order,)`, where $order = (deg[0] + 1) * (deg([1] + 1)$. The dtype will be the same as the converted *x* and *y*.

**See also:**

*polyvander*, *polyvander3d*, *polyval2d*, *polyval3d*

numpy.polynomial.polynomial.**polyvander3d**(*x*, *y*, *z*, *deg*)

Pseudo-Vandermonde matrix of given degrees.

Returns the pseudo-Vandermonde matrix of degrees *deg* and sample points *(x, y, z)*. If *l, m, n* are the given degrees in *x, y, z*, then The pseudo-Vandermonde matrix is defined by

$$V[..., (m + 1)(n + 1)i + (n + 1)j + k] = x^i * y^j * z^k,$$

where *0 <= i <= l*, *0 <= j <= m*, and *0 <= j <= n*. The leading indices of *V* index the points *(x, y, z)* and the last index encodes the powers of *x*, *y*, and *z*.

If `V = polyvander3d(x, y, z, [xdeg, ydeg, zdeg])`, then the columns of *V* correspond to the elements of a 3-D coefficient array *c* of shape (xdeg + 1, ydeg + 1, zdeg + 1) in the order

$$c_{000}, c_{001}, c_{002}, ..., c_{010}, c_{011}, c_{012}, \cdots$$

and `np.dot(V, c.flat)` and `polyval3d(x, y, z, c)` will be the same up to roundoff. This equivalence is useful both for least squares fitting and for the evaluation of a large number of 3-D polynomials of the same degrees and sample points.

**Parameters**

**x, y, z**

[array_like] Arrays of point coordinates, all of the same shape. The dtypes will be converted to either float64 or complex128 depending on whether any of the elements are complex. Scalars are converted to 1-D arrays.

**deg**

[list of ints] List of maximum degrees of the form [x_deg, y_deg, z_deg].

**Returns**

**vander3d**

[ndarray] The shape of the returned matrix is `x.shape + (order,)`, where $order = (deg[0] + 1) * (deg([1] + 1) * (deg[2] + 1)$. The dtype will be the same as the converted *x*, *y*, and *z*.

**See also:**

*polyvander*, *polyvander3d*, *polyval2d*, *polyval3d*

## Notes

New in version 1.7.0.

`numpy.polynomial.polynomial.`**`polycompanion`**(*c*)

Return the companion matrix of c.

The companion matrix for power series cannot be made symmetric by scaling the basis, so this function differs from those for the orthogonal polynomials.

**Parameters**

**c**

[array_like] 1-D array of polynomial coefficients ordered from low to high degree.

**Returns**

**mat**

[ndarray] Companion matrix of dimensions (deg, deg).

## Notes

New in version 1.7.0.

`numpy.polynomial.polynomial.`**`polyfit`**(*x*, *y*, *deg*, *rcond=None*, *full=False*, *w=None*)

Least-squares fit of a polynomial to data.

Return the coefficients of a polynomial of degree *deg* that is the least squares fit to the data values *y* given at points *x*. If *y* is 1-D the returned coefficients will also be 1-D. If *y* is 2-D multiple fits are done, one for each column of *y*, and the resulting coefficients are stored in the corresponding columns of a 2-D return. The fitted polynomial(s) are in the form

$$p(x) = c_0 + c_1 * x + ... + c_n * x^n,$$

where *n* is *deg*.

**Parameters**

**x**

[array_like, shape (*M*,)] x-coordinates of the *M* sample (data) points `(x[i], y[i])`.

**y**

[array_like, shape (*M*,) or (*M*, *K*)] y-coordinates of the sample points. Several sets of sample points sharing the same x-coordinates can be (independently) fit with one call to `polyfit` by passing in for *y* a 2-D array that contains one data set per column.

**deg**

[int or 1-D array_like] Degree(s) of the fitting polynomials. If *deg* is a single integer all terms up to and including the *deg*'th term are included in the fit. For NumPy versions >= 1.11.0 a list of integers specifying the degrees of the terms to include may be used instead.

**rcond**

[float, optional] Relative condition number of the fit. Singular values smaller than *rcond*, relative to the largest singular value, will be ignored. The default value is `len(x)*eps`, where *eps* is the relative precision of the platform's float type, about 2e-16 in most cases.

**full**

[bool, optional] Switch determining the nature of the return value. When `False` (the default) just the coefficients are returned; when `True`, diagnostic information from the singular value decomposition (used to solve the fit's matrix equation) is also returned.

**w**

[array_like, shape (*M*,), optional] Weights. If not None, the contribution of each point `(x[i],y[i])` to the fit is weighted by *w[i]*. Ideally the weights are chosen so that the errors of the products `w[i]*y[i]` all have the same variance. The default value is None.

New in version 1.5.0.

**Returns**

**coef**

[ndarray, shape (*deg* + 1,) or (*deg* + 1, *K*)] Polynomial coefficients ordered from low to high. If *y* was 2-D, the coefficients in column *k* of *coef* represent the polynomial fit to the data in *y*'s *k*-th column.

**[residuals, rank, singular_values, rcond]**

[list] These values are only returned if *full* = True

resid – sum of squared residuals of the least squares fit rank – the numerical rank of the scaled Vandermonde matrix sv – singular values of the scaled Vandermonde matrix rcond – value of *rcond*.

For more details, see *linalg.lstsq*.

**Raises**

**RankWarning**

Raised if the matrix in the least-squares fit is rank deficient. The warning is only raised if *full* == False. The warnings can be turned off by:

```
>>> import warnings
>>> warnings.simplefilter('ignore', np.RankWarning)
```

**See also:**

`chebfit`, `legfit`, `lagfit`, `hermfit`, `hermefit`

**`polyval`**

Evaluates a polynomial.

**`polyvander`**

Vandermonde matrix for powers.

**`linalg.lstsq`**

Computes a least-squares fit from the matrix.

**`scipy.interpolate.UnivariateSpline`**

Computes spline fits.

### Notes

The solution is the coefficients of the polynomial *p* that minimizes the sum of the weighted squared errors

$$E = \sum_j w_j^2 * |y_j - p(x_j)|^2,$$

where the $w_j$ are the weights. This problem is solved by setting up the (typically) over-determined matrix equation:

$$V(x) * c = w * y,$$

where *V* is the weighted pseudo Vandermonde matrix of *x*, *c* are the coefficients to be solved for, *w* are the weights, and *y* are the observed values. This equation is then solved using the singular value decomposition of *V*.

If some of the singular values of *V* are so small that they are neglected (and *full* == `False`), a *RankWarning* will be raised. This means that the coefficient values may be poorly determined. Fitting to a lower order polynomial will usually get rid of the warning (but may not be what you want, of course; if you have independent reason(s) for choosing the degree which isn't working, you may have to: a) reconsider those reasons, and/or b) reconsider the quality of your data). The *rcond* parameter can also be set to a value smaller than its default, but the resulting fit may be spurious and have large contributions from roundoff error.

Polynomial fits using double precision tend to "fail" at about (polynomial) degree 20. Fits using Chebyshev or Legendre series are generally better conditioned, but much can still depend on the distribution of the sample points and the smoothness of the data. If the quality of the fit is inadequate, splines may be a good alternative.

### Examples

```
>>> np.random.seed(123)
>>> from numpy.polynomial import polynomial as P
>>> x = np.linspace(-1,1,51) # x "data": [-1, -0.96, ..., 0.96, 1]
>>> y = x**3 - x + np.random.randn(len(x)) # x^3 - x + N(0,1) "noise"
>>> c, stats = P.polyfit(x,y,3,full=True)
>>> np.random.seed(123)
>>> c # c[0], c[2] should be approx. 0, c[1] approx. -1, c[3] approx. 1
array([ 0.01909725, -1.30598256, -0.00577963,  1.02644286]) # may vary
>>> stats # note the large SSR, explaining the rather poor results
 [array([ 38.06116253]), 4, array([ 1.38446749,  1.32119158,  0.50443316, # may␣
↪vary
          0.28853036]), 1.1324274851176597e-014]
```

Same thing without the added noise

```
>>> y = x**3 - x
>>> c, stats = P.polyfit(x,y,3,full=True)
>>> c # c[0], c[2] should be "very close to 0", c[1] ~= -1, c[3] ~= 1
array([-6.36925336e-18, -1.00000000e+00, -4.08053781e-16,  1.00000000e+00])
>>> stats # note the minuscule SSR
[array([  7.46346754e-31]), 4, array([ 1.38446749,  1.32119158, # may vary
           0.50443316,  0.28853036]), 1.1324274851176597e-014]
```

`numpy.polynomial.polynomial.`**`polytrim`**`(`*c*, *tol=0*`)`

Remove "small" "trailing" coefficients from a polynomial.

"Small" means "small in absolute value" and is controlled by the parameter *tol*; "trailing" means highest order coefficient(s), e.g., in `[0, 1, 1, 0, 0]` (which represents `0 + x + x**2 + 0*x**3 + 0*x**4`) both the 3-rd and 4-th order coefficients would be "trimmed."

**Parameters**

**c**

[array_like] 1-d array of coefficients, ordered from lowest order to highest.

**tol**

[number, optional] Trailing (i.e., highest order) elements with absolute value less than or equal to *tol* (default value is zero) are removed.

**Returns**

**trimmed**

[ndarray] 1-d array with trailing zeros removed. If the resulting series would be empty, a series containing a single zero is returned.

**Raises**

**ValueError**

If *tol* < 0

**See also:**

`trimseq`

## Examples

```
>>> from numpy.polynomial import polyutils as pu
>>> pu.trimcoef((0,0,3,0,5,0,0))
array([0.,  0.,  3.,  0.,  5.])
>>> pu.trimcoef((0,0,1e-3,0,1e-5,0,0),1e-3) # item == tol is trimmed
array([0.])
>>> i = complex(0,1) # works for complex
>>> pu.trimcoef((3e-4,1e-3*(1-i),5e-4,2e-5*(1+i)), 1e-3)
array([0.0003+0.j   , 0.001 -0.001j])
```

`numpy.polynomial.polynomial.`**`polyline`**(*off*, *scl*)

Returns an array representing a linear polynomial.

**Parameters**

**off, scl**

[scalars] The "y-intercept" and "slope" of the line, respectively.

**Returns**

**y**

[ndarray] This module's representation of the linear polynomial `off + scl*x`.

**See also:**

`chebline`

### Examples

```
>>> from numpy.polynomial import polynomial as P
>>> P.polyline(1,-1)
array([ 1, -1])
>>> P.polyval(1, P.polyline(1,-1)) # should be 0
0.0
```

### See Also

`numpy.polynomial`

New in version 1.4.0.

## Chebyshev Series (`numpy.polynomial.chebyshev`)

This module provides a number of objects (mostly functions) useful for dealing with Chebyshev series, including a *`Chebyshev`* class that encapsulates the usual arithmetic operations. (General information on how this module represents and works with such polynomials is in the docstring for its "parent" sub-package, `numpy.polynomial`).

### Classes

| | |
|---|---|
| *`Chebyshev`*(coef[, domain, window]) | A Chebyshev series class. |

**`class`** `numpy.polynomial.chebyshev.`**`Chebyshev`**(*coef*, *domain=None*, *window=None*)

A Chebyshev series class.

The Chebyshev class provides the standard Python numerical methods '+', '-', '*', '//', '%', 'divmod', '**', and '()' as well as the methods listed below.

**Parameters**

**coef**

[array_like] Chebyshev coefficients in order of increasing degree, i.e., `(1, 2, 3)` gives `1*T_0(x) + 2*T_1(x) + 3*T_2(x)`.

**domain**

[(2,) array_like, optional] Domain to use. The interval `[domain[0], domain[1]]` is mapped to the interval `[window[0], window[1]]` by shifting and scaling. The default value is [-1, 1].

**window**

[(2,) array_like, optional] Window, see `domain` for its use. The default value is [-1, 1].

New in version 1.6.0.

### Methods

| | |
|---|---|
| `__call__`(self, arg) | Call self as a function. |
| `basis`(deg[, domain, window]) | Series basis polynomial of degree *deg*. |
| `cast`(series[, domain, window]) | Convert series to series of this class. |
| `convert`(self[, domain, kind, window]) | Convert series to a different kind and/or domain and/or window. |
| `copy`(self) | Return a copy. |
| `cutdeg`(self, deg) | Truncate series to the given degree. |
| `degree`(self) | The degree of the series. |
| `deriv`(self[, m]) | Differentiate. |
| `fit`(x, y, deg[, domain, rcond, full, w, window]) | Least squares fit to data. |
| `fromroots`(roots[, domain, window]) | Return series instance that has the specified roots. |
| `has_samecoef`(self, other) | Check if coefficients match. |
| `has_samedomain`(self, other) | Check if domains match. |
| `has_sametype`(self, other) | Check if types match. |
| `has_samewindow`(self, other) | Check if windows match. |
| `identity`([domain, window]) | Identity function. |
| `integ`(self[, m, k, lbnd]) | Integrate. |
| `interpolate`(func, deg[, domain, args]) | Interpolate a function at the Chebyshev points of the first kind. |
| `linspace`(self[, n, domain]) | Return x, y values at equally spaced points in domain. |
| `mapparms`(self) | Return the mapping parameters. |
| `roots`(self) | Return the roots of the series polynomial. |
| `trim`(self[, tol]) | Remove trailing coefficients |
| `truncate`(self, size) | Truncate series to length *size*. |

method

`Chebyshev.`**`__call__`**(*self*, *arg*)

Call self as a function.

method

**`classmethod`** `Chebyshev.`**`basis`**(*deg*, *domain=None*, *window=None*)

Series basis polynomial of degree *deg*.

Returns the series representing the basis polynomial of degree *deg*.

New in version 1.7.0.

**Parameters**

**deg**

[int] Degree of the basis polynomial for the series. Must be >= 0.

**domain**

[{None, array_like}, optional] If given, the array must be of the form `[beg, end]`, where `beg` and `end` are the endpoints of the domain. If None is given then the class domain is used. The default is None.

**window**

[{None, array_like}, optional] If given, the resulting array must be if the form `[beg, end]`, where `beg` and `end` are the endpoints of the window. If None is given then the class window is used. The default is None.

**Returns**

**new_series**

[series] A series with the coefficient of the *deg* term set to one and all others zero.

method

**classmethod** `Chebyshev.`**`cast`**(*series*, *domain=None*, *window=None*)

Convert series to series of this class.

The *series* is expected to be an instance of some polynomial series of one of the types supported by by the numpy.polynomial module, but could be some other class that supports the convert method.

New in version 1.7.0.

**Parameters**

**series**

[series] The series instance to be converted.

**domain**

[{None, array_like}, optional] If given, the array must be of the form `[beg, end]`, where `beg` and `end` are the endpoints of the domain. If None is given then the class domain is used. The default is None.

**window**

[{None, array_like}, optional] If given, the resulting array must be if the form `[beg, end]`, where `beg` and `end` are the endpoints of the window. If None is given then the class window is used. The default is None.

**Returns**

**new_series**

[series] A series of the same kind as the calling class and equal to *series* when evaluated.

**See also:**

***`convert`***

similar instance method

method

`Chebyshev.`**`convert`**(*self*, *domain=None*, *kind=None*, *window=None*)

Convert series to a different kind and/or domain and/or window.

**Parameters**

**domain**

[array_like, optional] The domain of the converted series. If the value is None, the default domain of *kind* is used.

**kind**

[class, optional] The polynomial series type class to which the current instance should be converted. If kind is None, then the class of the current instance is used.

**window**

[array_like, optional] The window of the converted series. If the value is None, the default window of *kind* is used.

**Returns**

**new_series**

[series] The returned class can be of different type than the current instance and/or have a different domain and/or different window.

### Notes

Conversion between domains and class types can result in numerically ill defined series.

method

`Chebyshev.`**`copy`**`(`*self*`)`

Return a copy.

**Returns**

**new_series**

[series] Copy of self.

method

`Chebyshev.`**`cutdeg`**`(`*self*, *deg*`)`

Truncate series to the given degree.

Reduce the degree of the series to *deg* by discarding the high order terms. If *deg* is greater than the current degree a copy of the current series is returned. This can be useful in least squares where the coefficients of the high degree terms may be very small.

New in version 1.5.0.

**Parameters**

**deg**

[non-negative int] The series is reduced to degree *deg* by discarding the high order terms. The value of *deg* must be a non-negative integer.

**Returns**

**new_series**

[series] New instance of series with reduced degree.

method

`Chebyshev.`**`degree`**`(`*self*`)`

The degree of the series.

New in version 1.5.0.

**Returns**

**degree**

[int] Degree of the series, one less than the number of coefficients.

method

`Chebyshev.`**`deriv`**`(`*self*, *m=1*`)`

Differentiate.

Return a series instance of that is the derivative of the current series.

**Parameters**

**m**

[non-negative int] Find the derivative of order *m*.

**Returns**

**new_series**

[series] A new series representing the derivative. The domain is the same as the domain of the differentiated series.

method

**`classmethod`** `Chebyshev.`**`fit`**`(`*x*, *y*, *deg*, *domain=None*, *rcond=None*, *full=False*, *w=None*, *window=None*`)`

Least squares fit to data.

Return a series instance that is the least squares fit to the data *y* sampled at *x*. The domain of the returned instance can be specified and this will often result in a superior fit with less chance of ill conditioning.

**Parameters**

**x**

[array_like, shape (M,)] x-coordinates of the M sample points `(x[i], y[i])`.

**y**

[array_like, shape (M,) or (M, K)] y-coordinates of the sample points. Several data sets of sample points sharing the same x-coordinates can be fitted at once by passing in a 2D-array that contains one dataset per column.

**deg**

[int or 1-D array_like] Degree(s) of the fitting polynomials. If *deg* is a single integer all terms up to and including the *deg*'th term are included in the fit. For NumPy versions >= 1.11.0 a list of integers specifying the degrees of the terms to include may be used instead.

**domain**

[{None, [beg, end], []}, optional] Domain to use for the returned series. If `None`, then a minimal domain that covers the points *x* is chosen. If `[]` the class domain is used. The default value was the class domain in NumPy 1.4 and `None` in later versions. The `[]` option was added in numpy 1.5.0.

**rcond**

[float, optional] Relative condition number of the fit. Singular values smaller than this relative to the largest singular value will be ignored. The default value is len(x)*eps, where eps is the relative precision of the float type, about 2e-16 in most cases.

**full**

[bool, optional] Switch determining nature of return value. When it is False (the default) just the coefficients are returned, when True diagnostic information from the singular value decomposition is also returned.

**w**

[array_like, shape (M,), optional] Weights. If not None the contribution of each point `(x[i],y[i])` to the fit is weighted by *w[i]*. Ideally the weights are chosen so that the errors of the products `w[i]*y[i]` all have the same variance. The default value is None.

New in version 1.5.0.

**window**

[{[beg, end]}, optional] Window to use for the returned series. The default value is the default class domain

New in version 1.6.0.

**Returns**

**new_series**

[series] A series that represents the least squares fit to the data and has the domain and window specified in the call. If the coefficients for the unscaled and unshifted basis polynomials are of interest, do `new_series.convert().coef`.

**[resid, rank, sv, rcond]**

[list] These values are only returned if *full* = True

resid – sum of squared residuals of the least squares fit rank – the numerical rank of the scaled Vandermonde matrix sv – singular values of the scaled Vandermonde matrix rcond – value of *rcond*.

For more details, see *linalg.lstsq*.

method

**`classmethod`** `Chebyshev.`**`fromroots`**(*roots*, *domain=[]*, *window=None*)

Return series instance that has the specified roots.

Returns a series representing the product `(x - r[0])*(x - r[1])*...*(x - r[n-1])`, where `r` is a list of roots.

**Parameters**

**roots**

[array_like] List of roots.

**domain**

[{[], None, array_like}, optional] Domain for the resulting series. If None the domain is the interval from the smallest root to the largest. If [] the domain is the class domain. The default is [].

**window**

[{None, array_like}, optional] Window for the returned series. If None the class window is used. The default is None.

**Returns**

**new_series**

[series] Series with the specified roots.

method

`Chebyshev.`**`has_samecoef`**`(`*self*, *other*`)`

Check if coefficients match.

New in version 1.6.0.

**Parameters**

**other**

[class instance] The other class must have the `coef` attribute.

**Returns**

**bool**

[boolean] True if the coefficients are the same, False otherwise.

method

`Chebyshev.`**`has_samedomain`**`(`*self*, *other*`)`

Check if domains match.

New in version 1.6.0.

**Parameters**

**other**

[class instance] The other class must have the `domain` attribute.

**Returns**

**bool**

[boolean] True if the domains are the same, False otherwise.

method

`Chebyshev.`**`has_sametype`**`(`*self*, *other*`)`

Check if types match.

New in version 1.7.0.

**Parameters**

**other**

[object] Class instance.

**Returns**

**bool**

[boolean] True if other is same class as self

method

`Chebyshev.`**`has_samewindow`**`(`*self*, *other*`)`

Check if windows match.

New in version 1.6.0.

**Parameters**

**other**

[class instance] The other class must have the `window` attribute.

**Returns**

**bool**

[boolean] True if the windows are the same, False otherwise.

method

**`classmethod`** `Chebyshev.`**`identity`**`(`*domain=None*, *window=None*`)`

Identity function.

If `p` is the returned series, then `p(x) == x` for all values of x.

**Parameters**

**domain**

[{None, array_like}, optional] If given, the array must be of the form `[beg, end]`, where `beg` and `end` are the endpoints of the domain. If None is given then the class domain is used. The default is None.

**window**

[{None, array_like}, optional] If given, the resulting array must be if the form `[beg, end]`, where `beg` and `end` are the endpoints of the window. If None is given then the class window is used. The default is None.

**Returns**

**new_series**

[series] Series of representing the identity.

method

`Chebyshev.`**`integ`**`(`*self*, *m=1*, *k=[]*, *lbnd=None*`)`

Integrate.

Return a series instance that is the definite integral of the current series.

**Parameters**

**m**

[non-negative int] The number of integrations to perform.

**k**

[array_like] Integration constants. The first constant is applied to the first integration, the second to the second, and so on. The list of values must less than or equal to *m* in length and any missing values are set to zero.

**lbnd**

[Scalar] The lower bound of the definite integral.

**Returns**

**new_series**

[series] A new series representing the integral. The domain is the same as the domain of the integrated series.

method

**`classmethod`** `Chebyshev.`**`interpolate`**(*func*, *deg*, *domain=None*, *args=()*)

Interpolate a function at the Chebyshev points of the first kind.

Returns the series that interpolates *func* at the Chebyshev points of the first kind scaled and shifted to the `domain`. The resulting series tends to a minmax approximation of *func* when the function is continuous in the domain.

New in version 1.14.0.

**Parameters**

**func**

[function] The function to be interpolated. It must be a function of a single variable of the form `f(x, a, b, c...)`, where `a, b, c...` are extra arguments passed in the *args* parameter.

**deg**

[int] Degree of the interpolating polynomial.

**domain**

[{None, [beg, end]}, optional] Domain over which *func* is interpolated. The default is None, in which case the domain is [-1, 1].

**args**

[tuple, optional] Extra arguments to be used in the function call. Default is no extra arguments.

**Returns**

**polynomial**

[Chebyshev instance] Interpolating Chebyshev instance.

### Notes

See *numpy.polynomial.chebfromfunction* for more details.

method

`Chebyshev.`**`linspace`**(*self*, *n=100*, *domain=None*)

Return x, y values at equally spaced points in domain.

Returns the x, y values at *n* linearly spaced points across the domain. Here y is the value of the polynomial at the points x. By default the domain is the same as that of the series instance. This method is intended mostly as a plotting aid.

New in version 1.5.0.

**Parameters**

**n**

[int, optional] Number of point pairs to return. The default value is 100.

**domain**

[{None, array_like}, optional] If not None, the specified domain is used instead of that of the calling instance. It should be of the form `[beg,end]`. The default is None which case the class domain is used.

**Returns**

**x, y**

[ndarray] x is equal to linspace(self.domain[0], self.domain[1], n) and y is the series evaluated at element of x.

method

`Chebyshev.`**`mapparms`**(*self*)

Return the mapping parameters.

The returned values define a linear map `off + scl*x` that is applied to the input arguments before the series is evaluated. The map depends on the `domain` and `window`; if the current `domain` is equal to the `window` the resulting map is the identity. If the coefficients of the series instance are to be used by themselves outside this class, then the linear function must be substituted for the `x` in the standard representation of the base polynomials.

**Returns**

**off, scl**

[float or complex] The mapping function is defined by `off + scl*x`.

### Notes

If the current domain is the interval `[l1, r1]` and the window is `[l2, r2]`, then the linear mapping function `L` is defined by the equations:

```
L(l1) = l2
L(r1) = r2
```

method

`Chebyshev.`**`roots`**`(`*self*`)`

Return the roots of the series polynomial.

Compute the roots for the series. Note that the accuracy of the roots decrease the further outside the domain they lie.

**Returns**

**roots**

[ndarray] Array containing the roots of the series.

method

`Chebyshev.`**`trim`**`(`*self*, *tol=0*`)`

Remove trailing coefficients

Remove trailing coefficients until a coefficient is reached whose absolute value greater than *tol* or the beginning of the series is reached. If all the coefficients would be removed the series is set to `[0]`. A new series instance is returned with the new coefficients. The current instance remains unchanged.

**Parameters**

**tol**

[non-negative number.] All trailing coefficients less than *tol* will be removed.

**Returns**

**new_series**

[series] Contains the new set of coefficients.

method

`Chebyshev.`**`truncate`**`(`*self*, *size*`)`

Truncate series to length *size*.

Reduce the series to length *size* by discarding the high degree terms. The value of *size* must be a positive integer. This can be useful in least squares where the coefficients of the high degree terms may be very small.

**Parameters**

**size**

[positive int] The series is reduced to length *size* by discarding the high degree terms. The value of *size* must be a positive integer.

**Returns**

**new_series**

[series] New instance of series with truncated coefficients.

## Constants

| |
|---|
| *chebdomain* |
| *chebzero* |
| *chebone* |
| *chebx* |

`numpy.polynomial.chebyshev.`**`chebdomain = array([-1, 1])`**

`numpy.polynomial.chebyshev.`**`chebzero = array([0])`**

`numpy.polynomial.chebyshev.`**`chebone = array([1])`**

`numpy.polynomial.chebyshev.`**`chebx = array([0, 1])`**

## Arithmetic

| | |
|---|---|
| *chebadd*(c1, c2) | Add one Chebyshev series to another. |
| *chebsub*(c1, c2) | Subtract one Chebyshev series from another. |
| *chebmulx*(c) | Multiply a Chebyshev series by x. |
| *chebmul*(c1, c2) | Multiply one Chebyshev series by another. |
| *chebdiv*(c1, c2) | Divide one Chebyshev series by another. |
| *chebpow*(c, pow[, maxpower]) | Raise a Chebyshev series to a power. |
| *chebval*(x, c[, tensor]) | Evaluate a Chebyshev series at points x. |
| *chebval2d*(x, y, c) | Evaluate a 2-D Chebyshev series at points (x, y). |
| *chebval3d*(x, y, z, c) | Evaluate a 3-D Chebyshev series at points (x, y, z). |
| *chebgrid2d*(x, y, c) | Evaluate a 2-D Chebyshev series on the Cartesian product of x and y. |
| *chebgrid3d*(x, y, z, c) | Evaluate a 3-D Chebyshev series on the Cartesian product of x, y, and z. |

`numpy.polynomial.chebyshev.`**`chebadd`**`(`*c1*, *c2*`)`

Add one Chebyshev series to another.

Returns the sum of two Chebyshev series $c1 + c2$. The arguments are sequences of coefficients ordered from lowest order term to highest, i.e., [1,2,3] represents the series `T_0 + 2*T_1 + 3*T_2`.

**Parameters**

**c1, c2**

[array_like] 1-D arrays of Chebyshev series coefficients ordered from low to high.

**Returns**

**out**

[ndarray] Array representing the Chebyshev series of their sum.

**See also:**

*chebsub*, *chebmulx*, *chebmul*, *chebdiv*, *chebpow*

## Notes

Unlike multiplication, division, etc., the sum of two Chebyshev series is a Chebyshev series (without having to "reproject" the result onto the basis set) so addition, just like that of "standard" polynomials, is simply "component-wise."

## Examples

```
>>> from numpy.polynomial import chebyshev as C
>>> c1 = (1,2,3)
>>> c2 = (3,2,1)
>>> C.chebadd(c1,c2)
array([4., 4., 4.])
```

`numpy.polynomial.chebyshev.`**`chebsub`**`(`*c1*, *c2*`)`

Subtract one Chebyshev series from another.

Returns the difference of two Chebyshev series *c1* - *c2*. The sequences of coefficients are from lowest order term to highest, i.e., [1,2,3] represents the series `T_0 + 2*T_1 + 3*T_2`.

**Parameters**

**c1, c2**

[array_like] 1-D arrays of Chebyshev series coefficients ordered from low to high.

**Returns**

**out**

[ndarray] Of Chebyshev series coefficients representing their difference.

**See also:**

*chebadd*, *chebmulx*, *chebmul*, *chebdiv*, *chebpow*

## Notes

Unlike multiplication, division, etc., the difference of two Chebyshev series is a Chebyshev series (without having to "reproject" the result onto the basis set) so subtraction, just like that of "standard" polynomials, is simply "component-wise."

## Examples

```
>>> from numpy.polynomial import chebyshev as C
>>> c1 = (1,2,3)
>>> c2 = (3,2,1)
>>> C.chebsub(c1,c2)
array([-2.,  0.,  2.])
>>> C.chebsub(c2,c1) # -C.chebsub(c1,c2)
array([ 2.,  0., -2.])
```

`numpy.polynomial.chebyshev.`**`chebmulx`**`(`*c*`)`

Multiply a Chebyshev series by x.

Multiply the polynomial *c* by x, where x is the independent variable.

**Parameters**

**c**

[array_like] 1-D array of Chebyshev series coefficients ordered from low to high.

**Returns**

**out**

[ndarray] Array representing the result of the multiplication.

## Notes

New in version 1.5.0.

## Examples

```
>>> from numpy.polynomial import chebyshev as C
>>> C.chebmulx([1,2,3])
array([1. , 2.5, 1. , 1.5])
```

`numpy.polynomial.chebyshev.`**`chebmul`**`(`*c1*, *c2*`)`

Multiply one Chebyshev series by another.

Returns the product of two Chebyshev series *c1* * *c2*. The arguments are sequences of coefficients, from lowest order "term" to highest, e.g., [1,2,3] represents the series `T_0 + 2*T_1 + 3*T_2`.

**Parameters**

**c1, c2**

[array_like] 1-D arrays of Chebyshev series coefficients ordered from low to high.

**Returns**

**out**

[ndarray] Of Chebyshev series coefficients representing their product.

**See also:**

*`chebadd`*, *`chebsub`*, *`chebmulx`*, *`chebdiv`*, *`chebpow`*

### Notes

In general, the (polynomial) product of two C-series results in terms that are not in the Chebyshev polynomial basis set. Thus, to express the product as a C-series, it is typically necessary to "reproject" the product onto said basis set, which typically produces "unintuitive live" (but correct) results; see Examples section below.

### Examples

```
>>> from numpy.polynomial import chebyshev as C
>>> c1 = (1,2,3)
>>> c2 = (3,2,1)
>>> C.chebmul(c1,c2) # multiplication requires "reprojection"
array([  6.5,  12. ,  12. ,   4. ,   1.5])
```

`numpy.polynomial.chebyshev.`**`chebdiv`**`(`*c1*, *c2*`)`

Divide one Chebyshev series by another.

Returns the quotient-with-remainder of two Chebyshev series *c1* / *c2*. The arguments are sequences of coefficients from lowest order "term" to highest, e.g., [1,2,3] represents the series `T_0 + 2*T_1 + 3*T_2`.

**Parameters**

**c1, c2**

[array_like] 1-D arrays of Chebyshev series coefficients ordered from low to high.

**Returns**

**[quo, rem]**

[ndarrays] Of Chebyshev series coefficients representing the quotient and remainder.

**See also:**

*chebadd*, *chebsub*, `chemulx`, *chebmul*, *chebpow*

### Notes

In general, the (polynomial) division of one C-series by another results in quotient and remainder terms that are not in the Chebyshev polynomial basis set. Thus, to express these results as C-series, it is typically necessary to "reproject" the results onto said basis set, which typically produces "unintuitive" (but correct) results; see Examples section below.

### Examples

```
>>> from numpy.polynomial import chebyshev as C
>>> c1 = (1,2,3)
>>> c2 = (3,2,1)
>>> C.chebdiv(c1,c2) # quotient "intuitive," remainder not
(array([3.]), array([-8., -4.]))
>>> c2 = (0,1,2,3)
>>> C.chebdiv(c2,c1) # neither "intuitive"
(array([0., 2.]), array([-2., -4.]))
```

`numpy.polynomial.chebyshev.`**`chebpow`**(*c*, *pow*, *maxpower=16*)

Raise a Chebyshev series to a power.

Returns the Chebyshev series *c* raised to the power *pow*. The argument *c* is a sequence of coefficients ordered from low to high. i.e., [1,2,3] is the series `T_0 + 2*T_1 + 3*T_2.`

**Parameters**

**c**

[array_like] 1-D array of Chebyshev series coefficients ordered from low to high.

**pow**

[integer] Power to which the series will be raised

**maxpower**

[integer, optional] Maximum power allowed. This is mainly to limit growth of the series to unmanageable size. Default is 16

**Returns**

**coef**

[ndarray] Chebyshev series of power.

**See also:**

*chebadd*, *chebsub*, *chebmulx*, *chebmul*, *chebdiv*

### Examples

```
>>> from numpy.polynomial import chebyshev as C
>>> C.chebpow([1, 2, 3, 4], 2)
array([15.5, 22. , 16. , ..., 12.5, 12. ,  8. ])
```

`numpy.polynomial.chebyshev.`**`chebval`**(*x*, *c*, *tensor=True*)

Evaluate a Chebyshev series at points x.

If *c* is of length *n + 1*, this function returns the value:

$$p(x) = c_0 * T_0(x) + c_1 * T_1(x) + ... + c_n * T_n(x)$$

The parameter *x* is converted to an array only if it is a tuple or a list, otherwise it is treated as a scalar. In either case, either *x* or its elements must support multiplication and addition both with themselves and with the elements of *c*.

If *c* is a 1-D array, then *p(x)* will have the same shape as *x*. If *c* is multidimensional, then the shape of the result depends on the value of *tensor*. If *tensor* is true the shape will be c.shape[1:] + x.shape. If *tensor* is false the shape will be c.shape[1:]. Note that scalars have shape (,).

Trailing zeros in the coefficients will be used in the evaluation, so they should be avoided if efficiency is a concern.

**Parameters**

**x**

[array_like, compatible object] If *x* is a list or tuple, it is converted to an ndarray, otherwise it is left unchanged and treated as a scalar. In either case, *x* or its elements must support addition and multiplication with with themselves and with the elements of *c*.

**c**

[array_like] Array of coefficients ordered so that the coefficients for terms of degree n are contained in c[n]. If *c* is multidimensional the remaining indices enumerate multiple polynomials. In the two dimensional case the coefficients may be thought of as stored in the columns of *c*.

**tensor**

[boolean, optional] If True, the shape of the coefficient array is extended with ones on the right, one for each dimension of *x*. Scalars have dimension 0 for this action. The result is that every column of coefficients in *c* is evaluated for every element of *x*. If False, *x* is broadcast over the columns of *c* for the evaluation. This keyword is useful when *c* is multidimensional. The default value is True.

New in version 1.7.0.

**Returns**

**values**

[ndarray, algebra_like] The shape of the return value is described above.

**See also:**

*chebval2d*, *chebgrid2d*, *chebval3d*, *chebgrid3d*

## Notes

The evaluation uses Clenshaw recursion, aka synthetic division.

`numpy.polynomial.chebyshev.`**`chebval2d`**`(x, y, c)`

Evaluate a 2-D Chebyshev series at points (x, y).

This function returns the values:

$$p(x,y) = \sum_{i,j} c_{i,j} * T_i(x) * T_j(y)$$

The parameters *x* and *y* are converted to arrays only if they are tuples or a lists, otherwise they are treated as a scalars and they must have the same shape after conversion. In either case, either *x* and *y* or their elements must support multiplication and addition both with themselves and with the elements of *c*.

If *c* is a 1-D array a one is implicitly appended to its shape to make it 2-D. The shape of the result will be c.shape[2:] + x.shape.

**Parameters**

**x, y**

[array_like, compatible objects] The two dimensional series is evaluated at the points *(x, y)*, where *x* and *y* must have the same shape. If *x* or *y* is a list or tuple, it is first converted to an ndarray, otherwise it is left unchanged and if it isn't an ndarray it is treated as a scalar.

**c**

[array_like] Array of coefficients ordered so that the coefficient of the term of multi-degree i,j is contained in `c[i,j]`. If *c* has dimension greater than 2 the remaining indices enumerate multiple sets of coefficients.

**Returns**

**values**

[ndarray, compatible object] The values of the two dimensional Chebyshev series at points formed from pairs of corresponding values from *x* and *y*.

**See also:**

*chebval*, *chebgrid2d*, *chebval3d*, *chebgrid3d*

### Notes

New in version 1.7.0.

`numpy.polynomial.chebyshev.`**`chebval3d`**`(`*x*, *y*, *z*, *c*`)`

Evaluate a 3-D Chebyshev series at points (x, y, z).

This function returns the values:

$$p(x,y,z)=\sum_{i,j,k}c_{i,j,k}*T_i(x)*T_j(y)*T_k(z)$$

The parameters *x*, *y*, and *z* are converted to arrays only if they are tuples or a lists, otherwise they are treated as a scalars and they must have the same shape after conversion. In either case, either *x*, *y*, and *z* or their elements must support multiplication and addition both with themselves and with the elements of *c*.

If *c* has fewer than 3 dimensions, ones are implicitly appended to its shape to make it 3-D. The shape of the result will be c.shape[3:] + x.shape.

**Parameters**

**x, y, z**

[array_like, compatible object] The three dimensional series is evaluated at the points *(x, y, z)*, where *x*, *y*, and *z* must have the same shape. If any of *x*, *y*, or *z* is a list or tuple, it is first converted to an ndarray, otherwise it is left unchanged and if it isn't an ndarray it is treated as a scalar.

**c**

[array_like] Array of coefficients ordered so that the coefficient of the term of multi-degree i,j,k is contained in `c[i,j,k]`. If *c* has dimension greater than 3 the remaining indices enumerate multiple sets of coefficients.

**Returns**

**values**

[ndarray, compatible object] The values of the multidimensional polynomial on points formed with triples of corresponding values from *x*, *y*, and *z*.

**See also:**

*chebval*, *chebval2d*, *chebgrid2d*, *chebgrid3d*

## Notes

New in version 1.7.0.

`numpy.polynomial.chebyshev.`**`chebgrid2d`**(*x*, *y*, *c*)

Evaluate a 2-D Chebyshev series on the Cartesian product of x and y.

This function returns the values:

$$p(a,b) = \sum_{i,j} c_{i,j} * T_i(a) * T_j(b),$$

where the points *(a, b)* consist of all pairs formed by taking *a* from *x* and *b* from *y*. The resulting points form a grid with *x* in the first dimension and *y* in the second.

The parameters *x* and *y* are converted to arrays only if they are tuples or a lists, otherwise they are treated as a scalars. In either case, either *x* and *y* or their elements must support multiplication and addition both with themselves and with the elements of *c*.

If *c* has fewer than two dimensions, ones are implicitly appended to its shape to make it 2-D. The shape of the result will be c.shape[2:] + x.shape + y.shape.

**Parameters**

**x, y**

[array_like, compatible objects] The two dimensional series is evaluated at the points in the Cartesian product of *x* and *y*. If *x* or *y* is a list or tuple, it is first converted to an ndarray, otherwise it is left unchanged and, if it isn't an ndarray, it is treated as a scalar.

**c**

[array_like] Array of coefficients ordered so that the coefficient of the term of multi-degree i,j is contained in *c[i,j]*. If *c* has dimension greater than two the remaining indices enumerate multiple sets of coefficients.

**Returns**

**values**

[ndarray, compatible object] The values of the two dimensional Chebyshev series at points in the Cartesian product of *x* and *y*.

**See also:**

*`chebval`*, *`chebval2d`*, *`chebval3d`*, *`chebgrid3d`*

## Notes

New in version 1.7.0.

`numpy.polynomial.chebyshev.`**`chebgrid3d`**(*x*, *y*, *z*, *c*)

Evaluate a 3-D Chebyshev series on the Cartesian product of x, y, and z.

This function returns the values:

$$p(a,b,c) = \sum_{i,j,k} c_{i,j,k} * T_i(a) * T_j(b) * T_k(c)$$

where the points *(a, b, c)* consist of all triples formed by taking *a* from *x*, *b* from *y*, and *c* from *z*. The resulting points form a grid with *x* in the first dimension, *y* in the second, and *z* in the third.

The parameters *x*, *y*, and *z* are converted to arrays only if they are tuples or a lists, otherwise they are treated as a scalars. In either case, either *x*, *y*, and *z* or their elements must support multiplication and addition both with themselves and with the elements of *c*.

If *c* has fewer than three dimensions, ones are implicitly appended to its shape to make it 3-D. The shape of the result will be c.shape[3:] + x.shape + y.shape + z.shape.

**Parameters**

**x, y, z**

[array_like, compatible objects] The three dimensional series is evaluated at the points in the Cartesian product of *x*, *y*, and *z*. If *x*,'y', or *z* is a list or tuple, it is first converted to an ndarray, otherwise it is left unchanged and, if it isn't an ndarray, it is treated as a scalar.

**c**

[array_like] Array of coefficients ordered so that the coefficients for terms of degree i,j are contained in `c[i,j]`. If *c* has dimension greater than two the remaining indices enumerate multiple sets of coefficients.

**Returns**

**values**

[ndarray, compatible object] The values of the two dimensional polynomial at points in the Cartesian product of *x* and *y*.

**See also:**

*chebval*, *chebval2d*, *chebgrid2d*, *chebval3d*

### Notes

New in version 1.7.0.

## Calculus

| | |
|---|---|
| *chebder*(c[, m, scl, axis]) | Differentiate a Chebyshev series. |
| *chebint*(c[, m, k, lbnd, scl, axis]) | Integrate a Chebyshev series. |

`numpy.polynomial.chebyshev.`**`chebder`**(*c*, *m=1*, *scl=1*, *axis=0*)

Differentiate a Chebyshev series.

Returns the Chebyshev series coefficients *c* differentiated *m* times along *axis*. At each iteration the result is multiplied by *scl* (the scaling factor is for use in a linear change of variable). The argument *c* is an array of coefficients from low to high degree along each axis, e.g., [1,2,3] represents the series `1*T_0 + 2*T_1 + 3*T_2` while [[1,2],[1,2]] represents `1*T_0(x)*T_0(y) + 1*T_1(x)*T_0(y) + 2*T_0(x)*T_1(y) + 2*T_1(x)*T_1(y)` if axis=0 is `x` and axis=1 is `y`.

**Parameters**

**c**

[array_like] Array of Chebyshev series coefficients. If c is multidimensional the different axis correspond to different variables with the degree in each axis given by the corresponding index.

**m**

[int, optional] Number of derivatives taken, must be non-negative. (Default: 1)

**scl**

[scalar, optional] Each differentiation is multiplied by *scl*. The end result is multiplication by `scl**m`. This is for use in a linear change of variable. (Default: 1)

**axis**

[int, optional] Axis over which the derivative is taken. (Default: 0).

New in version 1.7.0.

**Returns**

**der**

[ndarray] Chebyshev series of the derivative.

**See also:**

`chebint`

### Notes

In general, the result of differentiating a C-series needs to be "reprojected" onto the C-series basis set. Thus, typically, the result of this function is "unintuitive," albeit correct; see Examples section below.

### Examples

```
>>> from numpy.polynomial import chebyshev as C
>>> c = (1,2,3,4)
>>> C.chebder(c)
array([14., 12., 24.])
>>> C.chebder(c,3)
array([96.])
>>> C.chebder(c,scl=-1)
array([-14., -12., -24.])
>>> C.chebder(c,2,-1)
array([12.,  96.])
```

`numpy.polynomial.chebyshev.`**`chebint`**`(`*c*, *m=1*, *k=[]*, *lbnd=0*, *scl=1*, *axis=0*`)`

Integrate a Chebyshev series.

Returns the Chebyshev series coefficients *c* integrated *m* times from *lbnd* along *axis*. At each iteration the resulting series is **multiplied** by *scl* and an integration constant, *k*, is added. The scaling factor is for use in a linear change of variable. ("Buyer beware": note that, depending on what one is doing, one may want *scl* to be the reciprocal of what one might expect; for more information, see the Notes section below.) The argument *c* is an array of coefficients from low to high degree along each axis, e.g., [1,2,3] represents the series `T_0 + 2*T_1 + 3*T_2` while [[1,2],[1,2]] represents `1*T_0(x)*T_0(y) + 1*T_1(x)*T_0(y) + 2*T_0(x)*T_1(y) + 2*T_1(x)*T_1(y)` if axis=0 is `x` and axis=1 is `y`.

**Parameters**

**c**

[array_like] Array of Chebyshev series coefficients. If c is multidimensional the different axis correspond to different variables with the degree in each axis given by the corresponding index.

**m**

[int, optional] Order of integration, must be positive. (Default: 1)

**k**

[{[], list, scalar}, optional] Integration constant(s). The value of the first integral at zero is the first value in the list, the value of the second integral at zero is the second value, etc. If `k == []` (the default), all constants are set to zero. If `m == 1`, a single scalar can be given instead of a list.

**lbnd**

[scalar, optional] The lower bound of the integral. (Default: 0)

**scl**

[scalar, optional] Following each integration the result is *multiplied* by *scl* before the integration constant is added. (Default: 1)

**axis**

[int, optional] Axis over which the integral is taken. (Default: 0).

New in version 1.7.0.

**Returns**

**S**

[ndarray] C-series coefficients of the integral.

**Raises**

**ValueError**

If `m < 1`, `len(k) > m`, `np.ndim(lbnd) != 0`, or `np.ndim(scl) != 0`.

**See also:**

*chebder*

### Notes

Note that the result of each integration is *multiplied* by *scl*. Why is this important to note? Say one is making a linear change of variable $u = ax + b$ in an integral relative to $x$. Then $dx = du/a$, so one will need to set *scl* equal to $1/a$- perhaps not what one would have first thought.

Also note that, in general, the result of integrating a C-series needs to be "reprojected" onto the C-series basis set. Thus, typically, the result of this function is "unintuitive," albeit correct; see Examples section below.

### Examples

```
>>> from numpy.polynomial import chebyshev as C
>>> c = (1,2,3)
>>> C.chebint(c)
array([ 0.5, -0.5,  0.5,  0.5])
>>> C.chebint(c,3)
array([ 0.03125   , -0.1875    ,  0.04166667, -0.05208333,  0.01041667, # may vary
    0.00625   ])
>>> C.chebint(c, k=3)
array([ 3.5, -0.5,  0.5,  0.5])
>>> C.chebint(c,lbnd=-2)
array([ 8.5, -0.5,  0.5,  0.5])
>>> C.chebint(c,scl=-2)
array([-1.,  1., -1., -1.])
```

## Misc Functions

| | |
|---|---|
| *chebfromroots*(roots) | Generate a Chebyshev series with given roots. |
| *chebroots*(c) | Compute the roots of a Chebyshev series. |
| *chebvander*(x, deg) | Pseudo-Vandermonde matrix of given degree. |
| *chebvander2d*(x, y, deg) | Pseudo-Vandermonde matrix of given degrees. |
| *chebvander3d*(x, y, z, deg) | Pseudo-Vandermonde matrix of given degrees. |
| *chebgauss*(deg) | Gauss-Chebyshev quadrature. |
| *chebweight*(x) | The weight function of the Chebyshev polynomials. |
| *chebcompanion*(c) | Return the scaled companion matrix of c. |
| *chebfit*(x, y, deg[, rcond, full, w]) | Least squares fit of Chebyshev series to data. |
| *chebpts1*(npts) | Chebyshev points of the first kind. |
| *chebpts2*(npts) | Chebyshev points of the second kind. |
| *chebtrim*(c[, tol]) | Remove “small” “trailing” coefficients from a polynomial. |
| *chebline*(off, scl) | Chebyshev series whose graph is a straight line. |
| *cheb2poly*(c) | Convert a Chebyshev series to a polynomial. |
| *poly2cheb*(pol) | Convert a polynomial to a Chebyshev series. |
| *chebinterpolate*(func, deg[, args]) | Interpolate a function at the Chebyshev points of the first kind. |

`numpy.polynomial.chebyshev.`**`chebfromroots`**`(`*roots*`)`

Generate a Chebyshev series with given roots.

The function returns the coefficients of the polynomial

$$p(x) = (x - r_0) * (x - r_1) * ... * (x - r_n),$$

in Chebyshev form, where the *r_n* are the roots specified in *roots*. If a zero has multiplicity n, then it must appear in *roots* n times. For instance, if 2 is a root of multiplicity three and 3 is a root of multiplicity 2, then *roots* looks something like [2, 2, 2, 3, 3]. The roots can appear in any order.

If the returned coefficients are *c*, then

$$p(x) = c_0 + c_1 * T_1(x) + ... + c_n * T_n(x)$$

The coefficient of the last term is not generally 1 for monic polynomials in Chebyshev form.

**Parameters**

**roots**

[array_like] Sequence containing the roots.

**Returns**

**out**

[ndarray] 1-D array of coefficients. If all roots are real then *out* is a real array, if some of the roots are complex, then *out* is complex even if all the coefficients in the result are real (see Examples below).

**See also:**

`polyfromroots`, `legfromroots`, `lagfromroots`, `hermfromroots`, `hermefromroots`

## Examples

```
>>> import numpy.polynomial.chebyshev as C
>>> C.chebfromroots((-1,0,1)) # x^3 - x relative to the standard basis
array([ 0.  , -0.25,  0.  ,  0.25])
>>> j = complex(0,1)
>>> C.chebfromroots((-j,j)) # x^2 + 1 relative to the standard basis
array([1.5+0.j, 0. +0.j, 0.5+0.j])
```

`numpy.polynomial.chebyshev.`**`chebroots`**`(`*c*`)`

Compute the roots of a Chebyshev series.

Return the roots (a.k.a. “zeros”) of the polynomial

$$p(x) = \sum_i c[i] * T_i(x).$$

**Parameters**

**c**

[1-D array_like] 1-D array of coefficients.

**Returns**

**out**

[ndarray] Array of the roots of the series. If all the roots are real, then *out* is also real, otherwise it is complex.

**See also:**

`polyroots`, `legroots`, `lagroots`, `hermroots`, `hermeroots`

### Notes

The root estimates are obtained as the eigenvalues of the companion matrix, Roots far from the origin of the complex plane may have large errors due to the numerical instability of the series for such values. Roots with multiplicity greater than 1 will also show larger errors as the value of the series near such points is relatively insensitive to errors in the roots. Isolated roots near the origin can be improved by a few iterations of Newton's method.

The Chebyshev series basis polynomials aren't powers of *x* so the results of this function may seem unintuitive.

### Examples

```
>>> import numpy.polynomial.chebyshev as cheb
>>> cheb.chebroots((-1, 1,-1, 1)) # T3 - T2 + T1 - T0 has real roots
array([ -5.00000000e-01,   2.60860684e-17,   1.00000000e+00]) # may vary
```

`numpy.polynomial.chebyshev.`**`chebvander`**(*x*, *deg*)

Pseudo-Vandermonde matrix of given degree.

Returns the pseudo-Vandermonde matrix of degree *deg* and sample points *x*. The pseudo-Vandermonde matrix is defined by

$$V[..., i] = T_i(x),$$

where *0 <= i <= deg*. The leading indices of *V* index the elements of *x* and the last index is the degree of the Chebyshev polynomial.

If *c* is a 1-D array of coefficients of length *n + 1* and *V* is the matrix `V = chebvander(x, n)`, then `np.dot(V, c)` and `chebval(x, c)` are the same up to roundoff. This equivalence is useful both for least squares fitting and for the evaluation of a large number of Chebyshev series of the same degree and sample points.

**Parameters**

**x**

[array_like] Array of points. The dtype is converted to float64 or complex128 depending on whether any of the elements are complex. If *x* is scalar it is converted to a 1-D array.

**deg**

[int] Degree of the resulting matrix.

**Returns**

**vander**

[ndarray] The pseudo Vandermonde matrix. The shape of the returned matrix is `x.shape + (deg + 1,)`, where The last index is the degree of the corresponding Chebyshev polynomial. The dtype will be the same as the converted *x*.

`numpy.polynomial.chebyshev.`**`chebvander2d`**(*x*, *y*, *deg*)

Pseudo-Vandermonde matrix of given degrees.

Returns the pseudo-Vandermonde matrix of degrees *deg* and sample points *(x, y)*. The pseudo-Vandermonde matrix is defined by

$$V[..., (deg[1] + 1) * i + j] = T_i(x) * T_j(y),$$

where *0 <= i <= deg[0]* and *0 <= j <= deg[1]*. The leading indices of *V* index the points *(x, y)* and the last index encodes the degrees of the Chebyshev polynomials.

If `V = chebvander2d(x, y, [xdeg, ydeg])`, then the columns of *V* correspond to the elements of a 2-D coefficient array *c* of shape (xdeg + 1, ydeg + 1) in the order

$$c_{00}, c_{01}, c_{02} ..., c_{10}, c_{11}, c_{12} ...$$

and `np.dot(V, c.flat)` and `chebval2d(x, y, c)` will be the same up to roundoff. This equivalence is useful both for least squares fitting and for the evaluation of a large number of 2-D Chebyshev series of the same degrees and sample points.

**Parameters**

**x, y**

[array_like] Arrays of point coordinates, all of the same shape. The dtypes will be converted to either float64 or complex128 depending on whether any of the elements are complex. Scalars are converted to 1-D arrays.

**deg**

[list of ints] List of maximum degrees of the form [x_deg, y_deg].

**Returns**

**vander2d**

[ndarray] The shape of the returned matrix is `x.shape + (order,)`, where $order = (deg[0] + 1) * (deg[1] + 1)$. The dtype will be the same as the converted *x* and *y*.

**See also:**

*chebvander*, *chebvander3d*, *chebval2d*, *chebval3d*

## Notes

New in version 1.7.0.

`numpy.polynomial.chebyshev.`**`chebvander3d`**`(`*x*, *y*, *z*, *deg*`)`

Pseudo-Vandermonde matrix of given degrees.

Returns the pseudo-Vandermonde matrix of degrees *deg* and sample points *(x, y, z)*. If *l, m, n* are the given degrees in *x, y, z*, then The pseudo-Vandermonde matrix is defined by

$$V[..., (m+1)(n+1)i + (n+1)j + k] = T_i(x) * T_j(y) * T_k(z),$$

where *0 <= i <= l*, *0 <= j <= m*, and *0 <= j <= n*. The leading indices of *V* index the points *(x, y, z)* and the last index encodes the degrees of the Chebyshev polynomials.

If `V = chebvander3d(x, y, z, [xdeg, ydeg, zdeg])`, then the columns of *V* correspond to the elements of a 3-D coefficient array *c* of shape (xdeg + 1, ydeg + 1, zdeg + 1) in the order

$$c_{000}, c_{001}, c_{002}, ..., c_{010}, c_{011}, c_{012}, ...$$

and `np.dot(V, c.flat)` and `chebval3d(x, y, z, c)` will be the same up to roundoff. This equivalence is useful both for least squares fitting and for the evaluation of a large number of 3-D Chebyshev series of the same degrees and sample points.

**Parameters**

**x, y, z**

[array_like] Arrays of point coordinates, all of the same shape. The dtypes will be converted to either float64 or complex128 depending on whether any of the elements are complex. Scalars are converted to 1-D arrays.

**deg**

[list of ints] List of maximum degrees of the form [x_deg, y_deg, z_deg].

**Returns**

**vander3d**

[ndarray] The shape of the returned matrix is `x.shape + (order,)`, where $order = (deg[0] + 1) * (deg[1] + 1) * (deg[2] + 1)$. The dtype will be the same as the converted *x*, *y*, and *z*.

**See also:**

*chebvander*, *chebvander3d*, *chebval2d*, *chebval3d*

## Notes

New in version 1.7.0.

`numpy.polynomial.chebyshev.`**`chebgauss`**(*deg*)

Gauss-Chebyshev quadrature.

Computes the sample points and weights for Gauss-Chebyshev quadrature. These sample points and weights will correctly integrate polynomials of degree $2 * deg - 1$ or less over the interval $[-1, 1]$ with the weight function $f(x) = 1/\sqrt{1 - x^2}$.

**Parameters**

**deg**

[int] Number of sample points and weights. It must be >= 1.

**Returns**

**x**

[ndarray] 1-D ndarray containing the sample points.

**y**

[ndarray] 1-D ndarray containing the weights.

### Notes

New in version 1.7.0.

The results have only been tested up to degree 100, higher degrees may be problematic. For Gauss-Chebyshev there are closed form solutions for the sample points and weights. If n = *deg*, then

$$x_i = \cos(\pi(2i - 1)/(2n))$$

$$w_i = \pi/n$$

`numpy.polynomial.chebyshev.`**`chebweight`**(*x*)

The weight function of the Chebyshev polynomials.

The weight function is $1/\sqrt{1 - x^2}$ and the interval of integration is $[-1, 1]$. The Chebyshev polynomials are orthogonal, but not normalized, with respect to this weight function.

**Parameters**

**x**

[array_like] Values at which the weight function will be computed.

**Returns**

**w**

[ndarray] The weight function at *x*.

### Notes

New in version 1.7.0.

`numpy.polynomial.chebyshev.`**`chebcompanion`**(*c*)

Return the scaled companion matrix of c.

The basis polynomials are scaled so that the companion matrix is symmetric when *c* is a Chebyshev basis polynomial. This provides better eigenvalue estimates than the unscaled case and for basis polynomials the eigenvalues are guaranteed to be real if `numpy.linalg.eigvalsh` is used to obtain them.

**Parameters**

**c**

[array_like] 1-D array of Chebyshev series coefficients ordered from low to high degree.

**Returns**

**mat**

[ndarray] Scaled companion matrix of dimensions (deg, deg).

### Notes

New in version 1.7.0.

`numpy.polynomial.chebyshev.`**`chebfit`**(*x*, *y*, *deg*, *rcond=None*, *full=False*, *w=None*)

Least squares fit of Chebyshev series to data.

Return the coefficients of a Chebyshev series of degree *deg* that is the least squares fit to the data values *y* given at points *x*. If *y* is 1-D the returned coefficients will also be 1-D. If *y* is 2-D multiple fits are done, one for each column of *y*, and the resulting coefficients are stored in the corresponding columns of a 2-D return. The fitted polynomial(s) are in the form

$$p(x) = c_0 + c_1 * T_1(x) + ... + c_n * T_n(x),$$

where *n* is *deg*.

**Parameters**

**x**

[array_like, shape (M,)] x-coordinates of the M sample points `(x[i], y[i])`.

**y**

[array_like, shape (M,) or (M, K)] y-coordinates of the sample points. Several data sets of sample points sharing the same x-coordinates can be fitted at once by passing in a 2D-array that contains one dataset per column.

**deg**

[int or 1-D array_like] Degree(s) of the fitting polynomials. If *deg* is a single integer, all terms up to and including the *deg*'th term are included in the fit. For NumPy versions >= 1.11.0 a list of integers specifying the degrees of the terms to include may be used instead.

**rcond**

[float, optional] Relative condition number of the fit. Singular values smaller than this relative to the largest singular value will be ignored. The default value is len(x)*eps, where eps is the relative precision of the float type, about 2e-16 in most cases.

**full**

[bool, optional] Switch determining nature of return value. When it is False (the default) just the coefficients are returned, when True diagnostic information from the singular value decomposition is also returned.

**w**

[array_like, shape (*M*,), optional] Weights. If not None, the contribution of each point `(x[i],y[i])` to the fit is weighted by *w[i]*. Ideally the weights are chosen so that the errors of the products `w[i]*y[i]` all have the same variance. The default value is None.

New in version 1.5.0.

**Returns**

**coef**

[ndarray, shape (M,) or (M, K)] Chebyshev coefficients ordered from low to high. If *y* was 2-D, the coefficients for the data in column k of *y* are in column *k*.

**[residuals, rank, singular_values, rcond]**

[list] These values are only returned if *full* = True

resid – sum of squared residuals of the least squares fit rank – the numerical rank of the scaled Vandermonde matrix sv – singular values of the scaled Vandermonde matrix rcond – value of *rcond*.

For more details, see *linalg.lstsq*.

**Warns**

**RankWarning**

The rank of the coefficient matrix in the least-squares fit is deficient. The warning is only raised if *full* = False. The warnings can be turned off by

```
>>> import warnings
>>> warnings.simplefilter('ignore', np.RankWarning)
```

**See also:**

`polyfit`, `legfit`, `lagfit`, `hermfit`, `hermefit`

**`chebval`**

Evaluates a Chebyshev series.

**`chebvander`**

Vandermonde matrix of Chebyshev series.

**`chebweight`**

Chebyshev weight function.

**`linalg.lstsq`**

Computes a least-squares fit from the matrix.

**`scipy.interpolate.UnivariateSpline`**

Computes spline fits.

## Notes

The solution is the coefficients of the Chebyshev series *p* that minimizes the sum of the weighted squared errors

$$E = \sum_j w_j^2 * |y_j - p(x_j)|^2,$$

where $w_j$ are the weights. This problem is solved by setting up as the (typically) overdetermined matrix equation

$$V(x) * c = w * y,$$

where *V* is the weighted pseudo Vandermonde matrix of *x*, *c* are the coefficients to be solved for, *w* are the weights, and *y* are the observed values. This equation is then solved using the singular value decomposition of *V*.

If some of the singular values of *V* are so small that they are neglected, then a *RankWarning* will be issued. This means that the coefficient values may be poorly determined. Using a lower order fit will usually get rid of the warning. The *rcond* parameter can also be set to a value smaller than its default, but the resulting fit may be spurious and have large contributions from roundoff error.

Fits using Chebyshev series are usually better conditioned than fits using power series, but much can depend on the distribution of the sample points and the smoothness of the data. If the quality of the fit is inadequate splines may be a good alternative.

## References

[1]

`numpy.polynomial.chebyshev.`**`chebpts1`**`(`*npts*`)`

Chebyshev points of the first kind.

The Chebyshev points of the first kind are the points `cos(x)`, where `x = [pi*(k + .5)/npts for k in range(npts)]`.

**Parameters**

**npts**

[int] Number of sample points desired.

**Returns**

**pts**

[ndarray] The Chebyshev points of the first kind.

**See also:**

*`chebpts2`*

## Notes

New in version 1.5.0.

`numpy.polynomial.chebyshev.`**`chebpts2`**`(`*npts*`)`

Chebyshev points of the second kind.

The Chebyshev points of the second kind are the points `cos(x)`, where `x = [pi*k/(npts - 1) for k in range(npts)]`.

**Parameters**

**npts**

[int] Number of sample points desired.

**Returns**

**pts**

[ndarray] The Chebyshev points of the second kind.

## Notes

New in version 1.5.0.

`numpy.polynomial.chebyshev.`**`chebtrim`**`(`*c*, *tol=0*`)`

Remove "small" "trailing" coefficients from a polynomial.

"Small" means "small in absolute value" and is controlled by the parameter *tol*; "trailing" means highest order coefficient(s), e.g., in `[0, 1, 1, 0, 0]` (which represents `0 + x + x**2 + 0*x**3 + 0*x**4`) both the 3-rd and 4-th order coefficients would be "trimmed."

**Parameters**

**c**

[array_like] 1-d array of coefficients, ordered from lowest order to highest.

**tol**

[number, optional] Trailing (i.e., highest order) elements with absolute value less than or equal to *tol* (default value is zero) are removed.

**Returns**

**trimmed**

[ndarray] 1-d array with trailing zeros removed. If the resulting series would be empty, a series containing a single zero is returned.

**Raises**

**ValueError**

If *tol* < 0

**See also:**

`trimseq`

## Examples

```
>>> from numpy.polynomial import polyutils as pu
>>> pu.trimcoef((0,0,3,0,5,0,0))
array([0.,  0.,  3.,  0.,  5.])
>>> pu.trimcoef((0,0,1e-3,0,1e-5,0,0),1e-3) # item == tol is trimmed
array([0.])
>>> i = complex(0,1) # works for complex
>>> pu.trimcoef((3e-4,1e-3*(1-i),5e-4,2e-5*(1+i)), 1e-3)
array([0.0003+0.j   , 0.001 -0.001j])
```

`numpy.polynomial.chebyshev.`**`chebline`**`(`*off*, *scl*`)`

Chebyshev series whose graph is a straight line.

**Parameters**

**off, scl**

[scalars] The specified line is given by `off + scl*x`.

**Returns**

**y**

[ndarray] This module's representation of the Chebyshev series for `off + scl*x`.

**See also:**

`polyline`

## Examples

```
>>> import numpy.polynomial.chebyshev as C
>>> C.chebline(3,2)
array([3, 2])
>>> C.chebval(-3, C.chebline(3,2)) # should be -3
-3.0
```

`numpy.polynomial.chebyshev.`**`cheb2poly`**`(c)`

Convert a Chebyshev series to a polynomial.

Convert an array representing the coefficients of a Chebyshev series, ordered from lowest degree to highest, to an array of the coefficients of the equivalent polynomial (relative to the "standard" basis) ordered from lowest to highest degree.

**Parameters**

**c**

[array_like] 1-D array containing the Chebyshev series coefficients, ordered from lowest order term to highest.

**Returns**

**pol**

[ndarray] 1-D array containing the coefficients of the equivalent polynomial (relative to the "standard" basis) ordered from lowest order term to highest.

**See also:**

*`poly2cheb`*

## Notes

The easy way to do conversions between polynomial basis sets is to use the convert method of a class instance.

## Examples

```
>>> from numpy import polynomial as P
>>> c = P.Chebyshev(range(4))
>>> c
Chebyshev([0., 1., 2., 3.], domain=[-1,  1], window=[-1,  1])
>>> p = c.convert(kind=P.Polynomial)
>>> p
Polynomial([-2., -8.,  4., 12.], domain=[-1.,  1.], window=[-1.,  1.])
>>> P.chebyshev.cheb2poly(range(4))
array([-2.,  -8.,   4.,  12.])
```

`numpy.polynomial.chebyshev.`**`poly2cheb`**(*pol*)

Convert a polynomial to a Chebyshev series.

Convert an array representing the coefficients of a polynomial (relative to the "standard" basis) ordered from lowest degree to highest, to an array of the coefficients of the equivalent Chebyshev series, ordered from lowest to highest degree.

**Parameters**

**pol**

[array_like] 1-D array containing the polynomial coefficients

**Returns**

**c**

[ndarray] 1-D array containing the coefficients of the equivalent Chebyshev series.

**See also:**

*`cheb2poly`*

## Notes

The easy way to do conversions between polynomial basis sets is to use the convert method of a class instance.

## Examples

```
>>> from numpy import polynomial as P
>>> p = P.Polynomial(range(4))
>>> p
Polynomial([0., 1., 2., 3.], domain=[-1,  1], window=[-1,  1])
>>> c = p.convert(kind=P.Chebyshev)
>>> c
Chebyshev([1.  , 3.25, 1.  , 0.75], domain=[-1.,  1.], window=[-1.,  1.])
>>> P.chebyshev.poly2cheb(range(4))
array([1.  , 3.25, 1.  , 0.75])
```

`numpy.polynomial.chebyshev.`**`chebinterpolate`**(*func*, *deg*, *args=()*)

Interpolate a function at the Chebyshev points of the first kind.

Returns the Chebyshev series that interpolates *func* at the Chebyshev points of the first kind in the interval [-1, 1]. The interpolating series tends to a minmax approximation to *func* with increasing *deg* if the function is continuous in the interval.

New in version 1.14.0.

**Parameters**

**func**

[function] The function to be approximated. It must be a function of a single variable of the form `f(x, a, b, c...)`, where `a, b, c...` are extra arguments passed in the *args* parameter.

**deg**

[int] Degree of the interpolating polynomial

**args**

[tuple, optional] Extra arguments to be used in the function call. Default is no extra arguments.

**Returns**

**coef**

[ndarray, shape (deg + 1,)] Chebyshev coefficients of the interpolating series ordered from low to high.

### Notes

The Chebyshev polynomials used in the interpolation are orthogonal when sampled at the Chebyshev points of the first kind. If it is desired to constrain some of the coefficients they can simply be set to the desired value after the interpolation, no new interpolation or fit is needed. This is especially useful if it is known apriori that some of coefficients are zero. For instance, if the function is even then the coefficients of the terms of odd degree in the result can be set to zero.

### Examples

```
>>> import numpy.polynomial.chebyshev as C
>>> C.chebfromfunction(lambda x: np.tanh(x) + 0.5, 8)
array([  5.00000000e-01,   8.11675684e-01,  -9.86864911e-17,
        -5.42457905e-02,  -2.71387850e-16,   4.51658839e-03,
         2.46716228e-17,  -3.79694221e-04,  -3.26899002e-16])
```

## See also

`numpy.polynomial`

### Notes

The implementations of multiplication, division, integration, and differentiation use the algebraic identities [1]:

$$T_n(x) = \frac{z^n + z^{-n}}{2}$$
$$z\frac{dx}{dz} = \frac{z - z^{-1}}{2}.$$

where

$$x = \frac{z + z^{-1}}{2}.$$

These identities allow a Chebyshev series to be expressed as a finite, symmetric Laurent series. In this module, this sort of Laurent series is referred to as a "z-series."

### References

New in version 1.6.0.

## Hermite Series, "Physicists" (`numpy.polynomial.hermite`)

This module provides a number of objects (mostly functions) useful for dealing with Hermite series, including a `Hermite` class that encapsulates the usual arithmetic operations. (General information on how this module represents and works with such polynomials is in the docstring for its "parent" sub-package, `numpy.polynomial`).

### Classes

| | |
|---|---|
| `Hermite`(coef[, domain, window]) | An Hermite series class. |

**class** `numpy.polynomial.hermite.`**`Hermite`**(*coef*, *domain=None*, *window=None*)

An Hermite series class.

The Hermite class provides the standard Python numerical methods '+', '-', '*', '//', '%', 'divmod', '**', and '()' as well as the attributes and methods listed in the `ABCPolyBase` documentation.

**Parameters**

**coef**

[array_like] Hermite coefficients in order of increasing degree, i.e, `(1, 2, 3)` gives `1*H_0(x) + 2*H_1(X) + 3*H_2(x)`.

**domain**

[(2,) array_like, optional] Domain to use. The interval `[domain[0], domain[1]]` is mapped to the interval `[window[0], window[1]]` by shifting and scaling. The default value is [-1, 1].

**window**

[(2,) array_like, optional] Window, see `domain` for its use. The default value is [-1, 1].

New in version 1.6.0.

## Methods

| | |
|---|---|
| `__call__`(self, arg) | Call self as a function. |
| `basis`(deg[, domain, window]) | Series basis polynomial of degree *deg*. |
| `cast`(series[, domain, window]) | Convert series to series of this class. |
| `convert`(self[, domain, kind, window]) | Convert series to a different kind and/or domain and/or window. |
| `copy`(self) | Return a copy. |
| `cutdeg`(self, deg) | Truncate series to the given degree. |
| `degree`(self) | The degree of the series. |
| `deriv`(self[, m]) | Differentiate. |
| `fit`(x, y, deg[, domain, rcond, full, w, window]) | Least squares fit to data. |
| `fromroots`(roots[, domain, window]) | Return series instance that has the specified roots. |
| `has_samecoef`(self, other) | Check if coefficients match. |
| `has_samedomain`(self, other) | Check if domains match. |
| `has_sametype`(self, other) | Check if types match. |
| `has_samewindow`(self, other) | Check if windows match. |
| `identity`([domain, window]) | Identity function. |
| `integ`(self[, m, k, lbnd]) | Integrate. |
| `linspace`(self[, n, domain]) | Return x, y values at equally spaced points in domain. |
| `mapparms`(self) | Return the mapping parameters. |
| `roots`(self) | Return the roots of the series polynomial. |
| `trim`(self[, tol]) | Remove trailing coefficients |
| `truncate`(self, size) | Truncate series to length *size*. |

method

`Hermite.`**`__call__`**(*self*, *arg*)

Call self as a function.

method

**`classmethod`** `Hermite.`**`basis`**(*deg*, *domain=None*, *window=None*)

Series basis polynomial of degree *deg*.

Returns the series representing the basis polynomial of degree *deg*.

New in version 1.7.0.

**Parameters**

**deg**

[int] Degree of the basis polynomial for the series. Must be >= 0.

**domain**

[{None, array_like}, optional] If given, the array must be of the form `[beg, end]`, where `beg` and `end` are the endpoints of the domain. If None is given then the class domain is used. The default is None.

**window**

[{None, array_like}, optional] If given, the resulting array must be if the form `[beg, end]`, where `beg` and `end` are the endpoints of the window. If None is given then the class window is used. The default is None.

**Returns**

**new_series**

[series] A series with the coefficient of the *deg* term set to one and all others zero.

method

**classmethod** `Hermite.`**`cast`**(*series*, *domain=None*, *window=None*)

Convert series to series of this class.

The *series* is expected to be an instance of some polynomial series of one of the types supported by by the numpy.polynomial module, but could be some other class that supports the convert method.

New in version 1.7.0.

**Parameters**

**series**

[series] The series instance to be converted.

**domain**

[{None, array_like}, optional] If given, the array must be of the form `[beg, end]`, where `beg` and `end` are the endpoints of the domain. If None is given then the class domain is used. The default is None.

**window**

[{None, array_like}, optional] If given, the resulting array must be if the form `[beg, end]`, where `beg` and `end` are the endpoints of the window. If None is given then the class window is used. The default is None.

**Returns**

**new_series**

[series] A series of the same kind as the calling class and equal to *series* when evaluated.

**See also:**

*`convert`*

similar instance method

method

`Hermite.`**`convert`**(*self*, *domain=None*, *kind=None*, *window=None*)

Convert series to a different kind and/or domain and/or window.

**Parameters**

**domain**

[array_like, optional] The domain of the converted series. If the value is None, the default domain of *kind* is used.

**kind**

[class, optional] The polynomial series type class to which the current instance should be converted. If kind is None, then the class of the current instance is used.

**window**

[array_like, optional] The window of the converted series. If the value is None, the default window of *kind* is used.

**Returns**

**new_series**

[series] The returned class can be of different type than the current instance and/or have a different domain and/or different window.

### Notes

Conversion between domains and class types can result in numerically ill defined series.

method

`Hermite.`**`copy`**`(`*self*`)`

Return a copy.

**Returns**

**new_series**

[series] Copy of self.

method

`Hermite.`**`cutdeg`**`(`*self*, *deg*`)`

Truncate series to the given degree.

Reduce the degree of the series to *deg* by discarding the high order terms. If *deg* is greater than the current degree a copy of the current series is returned. This can be useful in least squares where the coefficients of the high degree terms may be very small.

New in version 1.5.0.

**Parameters**

**deg**

[non-negative int] The series is reduced to degree *deg* by discarding the high order terms. The value of *deg* must be a non-negative integer.

**Returns**

**new_series**

[series] New instance of series with reduced degree.

method

`Hermite.`**`degree`**`(`*self*`)`

The degree of the series.

New in version 1.5.0.

**Returns**

**degree**

[int] Degree of the series, one less than the number of coefficients.

method

`Hermite.`**`deriv`**`(`*self*, *m=1*`)`

Differentiate.

Return a series instance of that is the derivative of the current series.

**Parameters**

**m**

[non-negative int] Find the derivative of order *m*.

**Returns**

**new_series**

[series] A new series representing the derivative. The domain is the same as the domain of the differentiated series.

method

**`classmethod`** `Hermite.`**`fit`**`(`*x*, *y*, *deg*, *domain=None*, *rcond=None*, *full=False*, *w=None*, *window=None*`)`

Least squares fit to data.

Return a series instance that is the least squares fit to the data *y* sampled at *x*. The domain of the returned instance can be specified and this will often result in a superior fit with less chance of ill conditioning.

**Parameters**

**x**

[array_like, shape (M,)] x-coordinates of the M sample points `(x[i], y[i])`.

**y**

[array_like, shape (M,) or (M, K)] y-coordinates of the sample points. Several data sets of sample points sharing the same x-coordinates can be fitted at once by passing in a 2D-array that contains one dataset per column.

**deg**

[int or 1-D array_like] Degree(s) of the fitting polynomials. If *deg* is a single integer all terms up to and including the *deg*'th term are included in the fit. For NumPy versions >= 1.11.0 a list of integers specifying the degrees of the terms to include may be used instead.

**domain**

[{None, [beg, end], []}, optional] Domain to use for the returned series. If `None`, then a minimal domain that covers the points *x* is chosen. If `[]` the class domain is used. The default value was the class domain in NumPy 1.4 and `None` in later versions. The `[]` option was added in numpy 1.5.0.

**rcond**

[float, optional] Relative condition number of the fit. Singular values smaller than this relative to the largest singular value will be ignored. The default value is len(x)*eps, where eps is the relative precision of the float type, about 2e-16 in most cases.

**full**

[bool, optional] Switch determining nature of return value. When it is False (the default) just the coefficients are returned, when True diagnostic information from the singular value decomposition is also returned.

**w**

[array_like, shape (M,), optional] Weights. If not None the contribution of each point `(x[i],y[i])` to the fit is weighted by *w[i]*. Ideally the weights are chosen so that the errors of the products `w[i]*y[i]` all have the same variance. The default value is None.

New in version 1.5.0.

**window**

[{[beg, end]}, optional] Window to use for the returned series. The default value is the default class domain

New in version 1.6.0.

**Returns**

**new_series**

[series] A series that represents the least squares fit to the data and has the domain and window specified in the call. If the coefficients for the unscaled and unshifted basis polynomials are of interest, do `new_series.convert().coef`.

**[resid, rank, sv, rcond]**

[list] These values are only returned if *full* = True

resid – sum of squared residuals of the least squares fit rank – the numerical rank of the scaled Vandermonde matrix sv – singular values of the scaled Vandermonde matrix rcond – value of *rcond*.

For more details, see *linalg.lstsq*.

method

**classmethod** `Hermite.`**fromroots**(*roots*, *domain=[]*, *window=None*)

Return series instance that has the specified roots.

Returns a series representing the product `(x - r[0])*(x - r[1])*...*(x - r[n-1])`, where `r` is a list of roots.

**Parameters**

**roots**

[array_like] List of roots.

**domain**

[{[], None, array_like}, optional] Domain for the resulting series. If None the domain is the interval from the smallest root to the largest. If [] the domain is the class domain. The default is [].

**window**

[{None, array_like}, optional] Window for the returned series. If None the class window is used. The default is None.

**Returns**

**new_series**

[series] Series with the specified roots.

method

`Hermite.`**`has_samecoef`**`(`*self*, *other*`)`

Check if coefficients match.

New in version 1.6.0.

**Parameters**

**other**

[class instance] The other class must have the `coef` attribute.

**Returns**

**bool**

[boolean] True if the coefficients are the same, False otherwise.

method

`Hermite.`**`has_samedomain`**`(`*self*, *other*`)`

Check if domains match.

New in version 1.6.0.

**Parameters**

**other**

[class instance] The other class must have the `domain` attribute.

**Returns**

**bool**

[boolean] True if the domains are the same, False otherwise.

method

`Hermite.`**`has_sametype`**`(`*self*, *other*`)`

Check if types match.

New in version 1.7.0.

**Parameters**

**other**

[object] Class instance.

**Returns**

**bool**

[boolean] True if other is same class as self

method

`Hermite.`**`has_samewindow`**`(`*self*, *other*`)`

Check if windows match.

New in version 1.6.0.

**Parameters**

**other**

[class instance] The other class must have the `window` attribute.

**Returns**

**bool**

[boolean] True if the windows are the same, False otherwise.

method

**`classmethod`** `Hermite.`**`identity`**`(`*domain=None*, *window=None*`)`

Identity function.

If `p` is the returned series, then `p(x) == x` for all values of x.

**Parameters**

**domain**

[{None, array_like}, optional] If given, the array must be of the form `[beg, end]`, where `beg` and `end` are the endpoints of the domain. If None is given then the class domain is used. The default is None.

**window**

[{None, array_like}, optional] If given, the resulting array must be if the form `[beg, end]`, where `beg` and `end` are the endpoints of the window. If None is given then the class window is used. The default is None.

**Returns**

**new_series**

[series] Series of representing the identity.

method

`Hermite.`**`integ`**`(`*self*, *m=1*, *k=[]*, *lbnd=None*`)`

Integrate.

Return a series instance that is the definite integral of the current series.

**Parameters**

**m**

[non-negative int] The number of integrations to perform.

**k**

[array_like] Integration constants. The first constant is applied to the first integration, the second to the second, and so on. The list of values must less than or equal to *m* in length and any missing values are set to zero.

**lbnd**

[Scalar] The lower bound of the definite integral.

**Returns**

**new_series**

[series] A new series representing the integral. The domain is the same as the domain of the integrated series.

method

`Hermite.`**`linspace`**(*self*, *n=100*, *domain=None*)

Return x, y values at equally spaced points in domain.

Returns the x, y values at *n* linearly spaced points across the domain. Here y is the value of the polynomial at the points x. By default the domain is the same as that of the series instance. This method is intended mostly as a plotting aid.

New in version 1.5.0.

**Parameters**

**n**

[int, optional] Number of point pairs to return. The default value is 100.

**domain**

[{None, array_like}, optional] If not None, the specified domain is used instead of that of the calling instance. It should be of the form `[beg,end]`. The default is None which case the class domain is used.

**Returns**

**x, y**

[ndarray] x is equal to linspace(self.domain[0], self.domain[1], n) and y is the series evaluated at element of x.

method

`Hermite.`**`mapparms`**(*self*)

Return the mapping parameters.

The returned values define a linear map `off + scl*x` that is applied to the input arguments before the series is evaluated. The map depends on the `domain` and `window`; if the current `domain` is equal to the `window` the resulting map is the identity. If the coefficients of the series instance are to be used by themselves outside this class, then the linear function must be substituted for the `x` in the standard representation of the base polynomials.

**Returns**

**off, scl**

[float or complex] The mapping function is defined by `off + scl*x`.

### Notes

If the current domain is the interval `[l1, r1]` and the window is `[l2, r2]`, then the linear mapping function `L` is defined by the equations:

```
L(l1) = l2
L(r1) = r2
```

method

`Hermite.`**`roots`**`(`*self*`)`

Return the roots of the series polynomial.

Compute the roots for the series. Note that the accuracy of the roots decrease the further outside the domain they lie.

**Returns**

**roots**

[ndarray] Array containing the roots of the series.

method

`Hermite.`**`trim`**`(`*self*, *tol=0*`)`

Remove trailing coefficients

Remove trailing coefficients until a coefficient is reached whose absolute value greater than *tol* or the beginning of the series is reached. If all the coefficients would be removed the series is set to `[0]`. A new series instance is returned with the new coefficients. The current instance remains unchanged.

**Parameters**

**tol**

[non-negative number.] All trailing coefficients less than *tol* will be removed.

**Returns**

**new_series**

[series] Contains the new set of coefficients.

method

`Hermite.`**`truncate`**`(`*self*, *size*`)`

Truncate series to length *size*.

Reduce the series to length *size* by discarding the high degree terms. The value of *size* must be a positive integer. This can be useful in least squares where the coefficients of the high degree terms may be very small.

**Parameters**

**size**

[positive int] The series is reduced to length *size* by discarding the high degree terms. The value of *size* must be a positive integer.

**Returns**

**new_series**

[series] New instance of series with truncated coefficients.

### Constants

| |
|---|
| *hermdomain* |
| *hermzero* |
| *hermone* |
| *hermx* |

`numpy.polynomial.hermite.`**`hermdomain = array([-1, 1])`**

`numpy.polynomial.hermite.`**`hermzero = array([0])`**

`numpy.polynomial.hermite.`**`hermone = array([1])`**

`numpy.polynomial.hermite.`**`hermx = array([0. , 0.5])`**

### Arithmetic

| | |
|---|---|
| *hermadd*(c1, c2) | Add one Hermite series to another. |
| *hermsub*(c1, c2) | Subtract one Hermite series from another. |
| *hermmulx*(c) | Multiply a Hermite series by x. |
| *hermmul*(c1, c2) | Multiply one Hermite series by another. |
| *hermdiv*(c1, c2) | Divide one Hermite series by another. |
| *hermpow*(c, pow[, maxpower]) | Raise a Hermite series to a power. |
| *hermval*(x, c[, tensor]) | Evaluate an Hermite series at points x. |
| *hermval2d*(x, y, c) | Evaluate a 2-D Hermite series at points (x, y). |
| *hermval3d*(x, y, z, c) | Evaluate a 3-D Hermite series at points (x, y, z). |
| *hermgrid2d*(x, y, c) | Evaluate a 2-D Hermite series on the Cartesian product of x and y. |
| *hermgrid3d*(x, y, z, c) | Evaluate a 3-D Hermite series on the Cartesian product of x, y, and z. |

`numpy.polynomial.hermite.`**`hermadd`**`(`*c1*, *c2*`)`

Add one Hermite series to another.

Returns the sum of two Hermite series *c1* + *c2*. The arguments are sequences of coefficients ordered from lowest order term to highest, i.e., [1,2,3] represents the series `P_0 + 2*P_1 + 3*P_2`.

**Parameters**

**c1, c2**

[array_like] 1-D arrays of Hermite series coefficients ordered from low to high.

**Returns**

**out**

[ndarray] Array representing the Hermite series of their sum.

**See also:**

*hermsub*, *hermmulx*, *hermmul*, *hermdiv*, *hermpow*

### Notes

Unlike multiplication, division, etc., the sum of two Hermite series is a Hermite series (without having to "reproject" the result onto the basis set) so addition, just like that of "standard" polynomials, is simply "component-wise."

### Examples

```
>>> from numpy.polynomial.hermite import hermadd
>>> hermadd([1, 2, 3], [1, 2, 3, 4])
array([2., 4., 6., 4.])
```

`numpy.polynomial.hermite.`**`hermsub`**`(`*c1*, *c2*`)`

Subtract one Hermite series from another.

Returns the difference of two Hermite series *c1* - *c2*. The sequences of coefficients are from lowest order term to highest, i.e., [1,2,3] represents the series `P_0 + 2*P_1 + 3*P_2`.

**Parameters**

**c1, c2**

[array_like] 1-D arrays of Hermite series coefficients ordered from low to high.

**Returns**

**out**

[ndarray] Of Hermite series coefficients representing their difference.

**See also:**

*hermadd*, *hermmulx*, *hermmul*, *hermdiv*, *hermpow*

### Notes

Unlike multiplication, division, etc., the difference of two Hermite series is a Hermite series (without having to "reproject" the result onto the basis set) so subtraction, just like that of "standard" polynomials, is simply "component-wise."

### Examples

```
>>> from numpy.polynomial.hermite import hermsub
>>> hermsub([1, 2, 3, 4], [1, 2, 3])
array([0.,  0.,  0.,  4.])
```

`numpy.polynomial.hermite.`**`hermmulx`**`(`*c*`)`

Multiply a Hermite series by x.

Multiply the Hermite series *c* by x, where x is the independent variable.

**Parameters**

**c**

[array_like] 1-D array of Hermite series coefficients ordered from low to high.

**Returns**

**out**

[ndarray] Array representing the result of the multiplication.

**See also:**

*hermadd*, *hermsub*, *hermmul*, *hermdiv*, *hermpow*

## Notes

The multiplication uses the recursion relationship for Hermite polynomials in the form

xP_i(x) = (P_{i + 1}(x)/2 + i*P_{i - 1}(x))

## Examples

```
>>> from numpy.polynomial.hermite import hermmulx
>>> hermmulx([1, 2, 3])
array([2. , 6.5, 1. , 1.5])
```

`numpy.polynomial.hermite.`**`hermmul`**`(`*c1*, *c2*`)`

Multiply one Hermite series by another.

Returns the product of two Hermite series *c1* * *c2*. The arguments are sequences of coefficients, from lowest order "term" to highest, e.g., [1,2,3] represents the series `P_0 + 2*P_1 + 3*P_2`.

**Parameters**

**c1, c2**

[array_like] 1-D arrays of Hermite series coefficients ordered from low to high.

**Returns**

**out**

[ndarray] Of Hermite series coefficients representing their product.

**See also:**

*hermadd*, *hermsub*, *hermmulx*, *hermdiv*, *hermpow*

## Notes

In general, the (polynomial) product of two C-series results in terms that are not in the Hermite polynomial basis set. Thus, to express the product as a Hermite series, it is necessary to "reproject" the product onto said basis set, which may produce "unintuitive" (but correct) results; see Examples section below.

## Examples

```
>>> from numpy.polynomial.hermite import hermmul
>>> hermmul([1, 2, 3], [0, 1, 2])
array([52.,  29.,  52.,   7.,   6.])
```

`numpy.polynomial.hermite.`**`hermdiv`**`(`*c1*, *c2*`)`

Divide one Hermite series by another.

Returns the quotient-with-remainder of two Hermite series *c1* / *c2*. The arguments are sequences of coefficients from lowest order "term" to highest, e.g., [1,2,3] represents the series `P_0 + 2*P_1 + 3*P_2`.

**Parameters**

**c1, c2**

[array_like] 1-D arrays of Hermite series coefficients ordered from low to high.

**Returns**

**[quo, rem]**

[ndarrays] Of Hermite series coefficients representing the quotient and remainder.

**See also:**

*`hermadd`*, *`hermsub`*, *`hermmulx`*, *`hermmul`*, *`hermpow`*

## Notes

In general, the (polynomial) division of one Hermite series by another results in quotient and remainder terms that are not in the Hermite polynomial basis set. Thus, to express these results as a Hermite series, it is necessary to "reproject" the results onto the Hermite basis set, which may produce "unintuitive" (but correct) results; see Examples section below.

## Examples

```
>>> from numpy.polynomial.hermite import hermdiv
>>> hermdiv([ 52.,  29.,  52.,   7.,   6.], [0, 1, 2])
(array([1., 2., 3.]), array([0.]))
>>> hermdiv([ 54.,  31.,  52.,   7.,   6.], [0, 1, 2])
(array([1., 2., 3.]), array([2., 2.]))
>>> hermdiv([ 53.,  30.,  52.,   7.,   6.], [0, 1, 2])
(array([1., 2., 3.]), array([1., 1.]))
```

`numpy.polynomial.hermite.`**`hermpow`**`(`*c*, *pow*, *maxpower=16*`)`

Raise a Hermite series to a power.

Returns the Hermite series *c* raised to the power *pow*. The argument *c* is a sequence of coefficients ordered from low to high. i.e., [1,2,3] is the series `P_0 + 2*P_1 + 3*P_2.`

**Parameters**

**c**

[array_like] 1-D array of Hermite series coefficients ordered from low to high.

**pow**

[integer] Power to which the series will be raised

**maxpower**

[integer, optional] Maximum power allowed. This is mainly to limit growth of the series to unmanageable size. Default is 16

**Returns**

**coef**

[ndarray] Hermite series of power.

**See also:**

`hermadd`, `hermsub`, `hermmulx`, `hermmul`, `hermdiv`

## Examples

```
>>> from numpy.polynomial.hermite import hermpow
>>> hermpow([1, 2, 3], 2)
array([81.,  52.,  82.,  12.,   9.])
```

`numpy.polynomial.hermite.`**`hermval`**(*x*, *c*, *tensor=True*)

Evaluate an Hermite series at points x.

If *c* is of length *n + 1*, this function returns the value:

$$p(x) = c_0 * H_0(x) + c_1 * H_1(x) + ... + c_n * H_n(x)$$

The parameter *x* is converted to an array only if it is a tuple or a list, otherwise it is treated as a scalar. In either case, either *x* or its elements must support multiplication and addition both with themselves and with the elements of *c*.

If *c* is a 1-D array, then *p(x)* will have the same shape as *x*. If *c* is multidimensional, then the shape of the result depends on the value of *tensor*. If *tensor* is true the shape will be c.shape[1:] + x.shape. If *tensor* is false the shape will be c.shape[1:]. Note that scalars have shape (,).

Trailing zeros in the coefficients will be used in the evaluation, so they should be avoided if efficiency is a concern.

**Parameters**

**x**

[array_like, compatible object] If *x* is a list or tuple, it is converted to an ndarray, otherwise it is left unchanged and treated as a scalar. In either case, *x* or its elements must support addition and multiplication with with themselves and with the elements of *c*.

**c**

[array_like] Array of coefficients ordered so that the coefficients for terms of degree n are contained in c[n]. If *c* is multidimensional the remaining indices enumerate multiple polynomials. In the two dimensional case the coefficients may be thought of as stored in the columns of *c*.

**tensor**

[boolean, optional] If True, the shape of the coefficient array is extended with ones on the right, one for each dimension of *x*. Scalars have dimension 0 for this action. The result is that every column of coefficients in *c* is evaluated for every element of *x*. If False, *x* is broadcast over the columns of *c* for the evaluation. This keyword is useful when *c* is multidimensional. The default value is True.

New in version 1.7.0.

**Returns**

**values**

[ndarray, algebra_like] The shape of the return value is described above.

**See also:**

*hermval2d*, *hermgrid2d*, *hermval3d*, *hermgrid3d*

## Notes

The evaluation uses Clenshaw recursion, aka synthetic division.

## Examples

```
>>> from numpy.polynomial.hermite import hermval
>>> coef = [1,2,3]
>>> hermval(1, coef)
11.0
>>> hermval([[1,2],[3,4]], coef)
array([[ 11.,   51.],
       [115.,  203.]])
```

`numpy.polynomial.hermite.`**`hermval2d`**`(`*x*, *y*, *c*`)`

Evaluate a 2-D Hermite series at points (x, y).

This function returns the values:

$$p(x,y) = \sum_{i,j} c_{i,j} * H_i(x) * H_j(y)$$

The parameters *x* and *y* are converted to arrays only if they are tuples or a lists, otherwise they are treated as a scalars and they must have the same shape after conversion. In either case, either *x* and *y* or their elements must support multiplication and addition both with themselves and with the elements of *c*.

If *c* is a 1-D array a one is implicitly appended to its shape to make it 2-D. The shape of the result will be c.shape[2:] + x.shape.

**Parameters**

**x, y**

[array_like, compatible objects] The two dimensional series is evaluated at the points *(x, y)*, where *x* and *y* must have the same shape. If *x* or *y* is a list or tuple, it is first converted to an ndarray, otherwise it is left unchanged and if it isn't an ndarray it is treated as a scalar.

**c**

[array_like] Array of coefficients ordered so that the coefficient of the term of multi-degree i,j is contained in `c[i,j]`. If *c* has dimension greater than two the remaining indices enumerate multiple sets of coefficients.

**Returns**

**values**

[ndarray, compatible object] The values of the two dimensional polynomial at points formed with pairs of corresponding values from *x* and *y*.

**See also:**

*hermval*, *hermgrid2d*, *hermval3d*, *hermgrid3d*

## Notes

New in version 1.7.0.

`numpy.polynomial.hermite.`**`hermval3d`**`(x, y, z, c)`

Evaluate a 3-D Hermite series at points (x, y, z).

This function returns the values:

$$p(x,y,z) = \sum_{i,j,k} c_{i,j,k} * H_i(x) * H_j(y) * H_k(z)$$

The parameters *x*, *y*, and *z* are converted to arrays only if they are tuples or a lists, otherwise they are treated as a scalars and they must have the same shape after conversion. In either case, either *x*, *y*, and *z* or their elements must support multiplication and addition both with themselves and with the elements of *c*.

If *c* has fewer than 3 dimensions, ones are implicitly appended to its shape to make it 3-D. The shape of the result will be c.shape[3:] + x.shape.

**Parameters**

**x, y, z**

[array_like, compatible object] The three dimensional series is evaluated at the points *(x, y, z)*, where *x*, *y*, and *z* must have the same shape. If any of *x*, *y*, or *z* is a list or tuple, it is first converted to an ndarray, otherwise it is left unchanged and if it isn't an ndarray it is treated as a scalar.

**c**

[array_like] Array of coefficients ordered so that the coefficient of the term of multi-degree i,j,k is contained in `c[i,j,k]`. If *c* has dimension greater than 3 the remaining indices enumerate multiple sets of coefficients.

**Returns**

**values**

[ndarray, compatible object] The values of the multidimensional polynomial on points formed with triples of corresponding values from *x*, *y*, and *z*.

**See also:**

*hermval*, *hermval2d*, *hermgrid2d*, *hermgrid3d*

## Notes

New in version 1.7.0.

`numpy.polynomial.hermite.`**`hermgrid2d`**`(x, y, c)`

Evaluate a 2-D Hermite series on the Cartesian product of x and y.

This function returns the values:

$$p(a,b) = \sum_{i,j} c_{i,j} * H_i(a) * H_j(b)$$

where the points *(a, b)* consist of all pairs formed by taking *a* from *x* and *b* from *y*. The resulting points form a grid with *x* in the first dimension and *y* in the second.

The parameters *x* and *y* are converted to arrays only if they are tuples or a lists, otherwise they are treated as a scalars. In either case, either *x* and *y* or their elements must support multiplication and addition both with themselves and with the elements of *c*.

If *c* has fewer than two dimensions, ones are implicitly appended to its shape to make it 2-D. The shape of the result will be c.shape[2:] + x.shape.

**Parameters**

**x, y**

[array_like, compatible objects] The two dimensional series is evaluated at the points in the Cartesian product of *x* and *y*. If *x* or *y* is a list or tuple, it is first converted to an ndarray, otherwise it is left unchanged and, if it isn't an ndarray, it is treated as a scalar.

**c**

[array_like] Array of coefficients ordered so that the coefficients for terms of degree i,j are contained in `c[i,j]`. If *c* has dimension greater than two the remaining indices enumerate multiple sets of coefficients.

**Returns**

**values**

[ndarray, compatible object] The values of the two dimensional polynomial at points in the Cartesian product of *x* and *y*.

**See also:**

*hermval*, *hermval2d*, *hermval3d*, *hermgrid3d*

## Notes

New in version 1.7.0.

`numpy.polynomial.hermite.`**`hermgrid3d`**`(`*x*, *y*, *z*, *c*`)`

Evaluate a 3-D Hermite series on the Cartesian product of x, y, and z.

This function returns the values:

$$p(a,b,c) = \sum_{i,j,k} c_{i,j,k} * H_i(a) * H_j(b) * H_k(c)$$

where the points *(a, b, c)* consist of all triples formed by taking *a* from *x*, *b* from *y*, and *c* from *z*. The resulting points form a grid with *x* in the first dimension, *y* in the second, and *z* in the third.

The parameters *x*, *y*, and *z* are converted to arrays only if they are tuples or a lists, otherwise they are treated as a scalars. In either case, either *x*, *y*, and *z* or their elements must support multiplication and addition both with themselves and with the elements of *c*.

If *c* has fewer than three dimensions, ones are implicitly appended to its shape to make it 3-D. The shape of the result will be c.shape[3:] + x.shape + y.shape + z.shape.

**Parameters**

**x, y, z**

[array_like, compatible objects] The three dimensional series is evaluated at the points in the Cartesian product of *x*, *y*, and *z*. If *x*,‘y‘, or *z* is a list or tuple, it is first converted to an ndarray, otherwise it is left unchanged and, if it isn’t an ndarray, it is treated as a scalar.

**c**

[array_like] Array of coefficients ordered so that the coefficients for terms of degree i,j are contained in `c[i,j]`. If *c* has dimension greater than two the remaining indices enumerate multiple sets of coefficients.

**Returns**

**values**

[ndarray, compatible object] The values of the two dimensional polynomial at points in the Cartesian product of *x* and *y*.

**See also:**

*`hermval`*, *`hermval2d`*, *`hermgrid2d`*, *`hermval3d`*

## Notes

New in version 1.7.0.

### Calculus

| | |
|---|---|
| *hermder*(c[, m, scl, axis]) | Differentiate a Hermite series. |
| *hermint*(c[, m, k, lbnd, scl, axis]) | Integrate a Hermite series. |

`numpy.polynomial.hermite.`**`hermder`**(*c*, *m=1*, *scl=1*, *axis=0*)

Differentiate a Hermite series.

Returns the Hermite series coefficients *c* differentiated *m* times along *axis*. At each iteration the result is multiplied by *scl* (the scaling factor is for use in a linear change of variable). The argument *c* is an array of coefficients from low to high degree along each axis, e.g., [1,2,3] represents the series `1*H_0 + 2*H_1 + 3*H_2` while [[1,2],[1,2]] represents `1*H_0(x)*H_0(y) + 1*H_1(x)*H_0(y) + 2*H_0(x)*H_1(y) + 2*H_1(x)*H_1(y)` if axis=0 is `x` and axis=1 is `y`.

**Parameters**

**c**

[array_like] Array of Hermite series coefficients. If *c* is multidimensional the different axis correspond to different variables with the degree in each axis given by the corresponding index.

**m**

[int, optional] Number of derivatives taken, must be non-negative. (Default: 1)

**scl**

[scalar, optional] Each differentiation is multiplied by *scl*. The end result is multiplication by `scl**m`. This is for use in a linear change of variable. (Default: 1)

**axis**

[int, optional] Axis over which the derivative is taken. (Default: 0).

New in version 1.7.0.

**Returns**

**der**

[ndarray] Hermite series of the derivative.

**See also:**

*hermint*

### Notes

In general, the result of differentiating a Hermite series does not resemble the same operation on a power series. Thus the result of this function may be "unintuitive," albeit correct; see Examples section below.

### Examples

```
>>> from numpy.polynomial.hermite import hermder
>>> hermder([ 1. ,  0.5,  0.5,  0.5])
array([1., 2., 3.])
>>> hermder([-0.5,  1./2.,  1./8.,  1./12.,  1./16.], m=2)
array([1., 2., 3.])
```

`numpy.polynomial.hermite.`**`hermint`**`(`*c*, *m=1*, *k=[]*, *lbnd=0*, *scl=1*, *axis=0*`)`

Integrate a Hermite series.

Returns the Hermite series coefficients *c* integrated *m* times from *lbnd* along *axis*. At each iteration the resulting series is **multiplied** by *scl* and an integration constant, *k*, is added. The scaling factor is for use in a linear change of variable. ("Buyer beware": note that, depending on what one is doing, one may want *scl* to be the reciprocal of what one might expect; for more information, see the Notes section below.) The argument *c* is an array of coefficients from low to high degree along each axis, e.g., [1,2,3] represents the series `H_0 + 2*H_1 + 3*H_2` while [[1,2],[1,2]] represents `1*H_0(x)*H_0(y) + 1*H_1(x)*H_0(y) + 2*H_0(x)*H_1(y) + 2*H_1(x)*H_1(y)` if axis=0 is `x` and axis=1 is `y`.

**Parameters**

**c**

[array_like] Array of Hermite series coefficients. If c is multidimensional the different axis correspond to different variables with the degree in each axis given by the corresponding index.

**m**

[int, optional] Order of integration, must be positive. (Default: 1)

**k**

[{[], list, scalar}, optional] Integration constant(s). The value of the first integral at `lbnd` is the first value in the list, the value of the second integral at `lbnd` is the second value, etc. If `k == []` (the default), all constants are set to zero. If `m == 1`, a single scalar can be given instead of a list.

**lbnd**

[scalar, optional] The lower bound of the integral. (Default: 0)

**scl**

[scalar, optional] Following each integration the result is *multiplied* by *scl* before the integration constant is added. (Default: 1)

**axis**

[int, optional] Axis over which the integral is taken. (Default: 0).

New in version 1.7.0.

**Returns**

**S**

[ndarray] Hermite series coefficients of the integral.

**Raises**

**ValueError**

If `m < 0`, `len(k) > m`, `np.ndim(lbnd) != 0`, or `np.ndim(scl) != 0`.

**See also:**

*hermder*

## Notes

Note that the result of each integration is *multiplied* by *scl*. Why is this important to note? Say one is making a linear change of variable $u = ax + b$ in an integral relative to $x$. Then $dx = du/a$, so one will need to set *scl* equal to $1/a$ - perhaps not what one would have first thought.

Also note that, in general, the result of integrating a C-series needs to be "reprojected" onto the C-series basis set. Thus, typically, the result of this function is "unintuitive," albeit correct; see Examples section below.

## Examples

```
>>> from numpy.polynomial.hermite import hermint
>>> hermint([1,2,3]) # integrate once, value 0 at 0.
array([1. , 0.5, 0.5, 0.5])
>>> hermint([1,2,3], m=2) # integrate twice, value & deriv 0 at 0
array([-0.5       ,  0.5       ,  0.125     ,  0.08333333,  0.0625    ]) # may␣
↪vary
>>> hermint([1,2,3], k=1) # integrate once, value 1 at 0.
array([2. , 0.5, 0.5, 0.5])
>>> hermint([1,2,3], lbnd=-1) # integrate once, value 0 at -1
array([-2. ,  0.5,  0.5,  0.5])
>>> hermint([1,2,3], m=2, k=[1,2], lbnd=-1)
array([ 1.66666667, -0.5       ,  0.125     ,  0.08333333,  0.0625    ]) # may␣
↪vary
```

## Misc Functions

| | |
|---|---|
| *hermfromroots*(roots) | Generate a Hermite series with given roots. |
| *hermroots*(c) | Compute the roots of a Hermite series. |
| *hermvander*(x, deg) | Pseudo-Vandermonde matrix of given degree. |
| *hermvander2d*(x, y, deg) | Pseudo-Vandermonde matrix of given degrees. |
| *hermvander3d*(x, y, z, deg) | Pseudo-Vandermonde matrix of given degrees. |
| *hermgauss*(deg) | Gauss-Hermite quadrature. |
| *hermweight*(x) | Weight function of the Hermite polynomials. |
| *hermcompanion*(c) | Return the scaled companion matrix of c. |
| *hermfit*(x, y, deg[, rcond, full, w]) | Least squares fit of Hermite series to data. |
| *hermtrim*(c[, tol]) | Remove "small" "trailing" coefficients from a polynomial. |
| *hermline*(off, scl) | Hermite series whose graph is a straight line. |
| *herm2poly*(c) | Convert a Hermite series to a polynomial. |
| *poly2herm*(pol) | Convert a polynomial to a Hermite series. |

`numpy.polynomial.hermite.`**`hermfromroots`**(*roots*)

Generate a Hermite series with given roots.

The function returns the coefficients of the polynomial

$$p(x) = (x - r_0) * (x - r_1) * ... * (x - r_n),$$

in Hermite form, where the *r_n* are the roots specified in *roots*. If a zero has multiplicity n, then it must appear in *roots* n times. For instance, if 2 is a root of multiplicity three and 3 is a root of multiplicity 2, then *roots* looks something like [2, 2, 2, 3, 3]. The roots can appear in any order.

If the returned coefficients are *c*, then

$$p(x) = c_0 + c_1 * H_1(x) + ... + c_n * H_n(x)$$

The coefficient of the last term is not generally 1 for monic polynomials in Hermite form.

**Parameters**

**roots**

[array_like] Sequence containing the roots.

**Returns**

**out**

[ndarray] 1-D array of coefficients. If all roots are real then *out* is a real array, if some of the roots are complex, then *out* is complex even if all the coefficients in the result are real (see Examples below).

**See also:**

`polyfromroots`, `legfromroots`, `lagfromroots`, `chebfromroots`, `hermefromroots`

## Examples

```
>>> from numpy.polynomial.hermite import hermfromroots, hermval
>>> coef = hermfromroots((-1, 0, 1))
>>> hermval((-1, 0, 1), coef)
array([0.,  0.,  0.])
>>> coef = hermfromroots((-1j, 1j))
>>> hermval((-1j, 1j), coef)
array([0.+0.j, 0.+0.j])
```

`numpy.polynomial.hermite.`**`hermroots`**`(c)`

Compute the roots of a Hermite series.

Return the roots (a.k.a. "zeros") of the polynomial

$$p(x) = \sum_i c[i] * H_i(x).$$

**Parameters**

**c**

[1-D array_like] 1-D array of coefficients.

**Returns**

**out**

[ndarray] Array of the roots of the series. If all the roots are real, then *out* is also real, otherwise it is complex.

**See also:**

`polyroots`, `legroots`, `lagroots`, `chebroots`, `hermeroots`

## Notes

The root estimates are obtained as the eigenvalues of the companion matrix, Roots far from the origin of the complex plane may have large errors due to the numerical instability of the series for such values. Roots with multiplicity greater than 1 will also show larger errors as the value of the series near such points is relatively insensitive to errors in the roots. Isolated roots near the origin can be improved by a few iterations of Newton's method.

The Hermite series basis polynomials aren't powers of *x* so the results of this function may seem unintuitive.

## Examples

```
>>> from numpy.polynomial.hermite import hermroots, hermfromroots
>>> coef = hermfromroots([-1, 0, 1])
>>> coef
array([0.   ,  0.25 ,  0.   ,  0.125])
>>> hermroots(coef)
array([-1.00000000e+00, -1.38777878e-17,  1.00000000e+00])
```

`numpy.polynomial.hermite.`**`hermvander`**(*x*, *deg*)

Pseudo-Vandermonde matrix of given degree.

Returns the pseudo-Vandermonde matrix of degree *deg* and sample points *x*. The pseudo-Vandermonde matrix is defined by

$$V[..., i] = H_i(x),$$

where *0 <= i <= deg*. The leading indices of *V* index the elements of *x* and the last index is the degree of the Hermite polynomial.

If *c* is a 1-D array of coefficients of length *n + 1* and *V* is the array `V = hermvander(x, n)`, then `np.dot(V, c)` and `hermval(x, c)` are the same up to roundoff. This equivalence is useful both for least squares fitting and for the evaluation of a large number of Hermite series of the same degree and sample points.

**Parameters**

**x**

[array_like] Array of points. The dtype is converted to float64 or complex128 depending on whether any of the elements are complex. If *x* is scalar it is converted to a 1-D array.

**deg**

[int] Degree of the resulting matrix.

**Returns**

**vander**

[ndarray] The pseudo-Vandermonde matrix. The shape of the returned matrix is `x.shape + (deg + 1,)`, where The last index is the degree of the corresponding Hermite polynomial. The dtype will be the same as the converted *x*.

## Examples

```
>>> from numpy.polynomial.hermite import hermvander
>>> x = np.array([-1, 0, 1])
>>> hermvander(x, 3)
array([[ 1., -2.,  2.,  4.],
       [ 1.,  0., -2., -0.],
       [ 1.,  2.,  2., -4.]])
```

`numpy.polynomial.hermite.`**`hermvander2d`**(*x*, *y*, *deg*)

Pseudo-Vandermonde matrix of given degrees.

Returns the pseudo-Vandermonde matrix of degrees *deg* and sample points *(x, y)*. The pseudo-Vandermonde matrix is defined by

$$V[..., (deg[1] + 1) * i + j] = H_i(x) * H_j(y),$$

where *0 <= i <= deg[0]* and *0 <= j <= deg[1]*. The leading indices of *V* index the points *(x, y)* and the last index encodes the degrees of the Hermite polynomials.

If `V = hermvander2d(x, y, [xdeg, ydeg])`, then the columns of *V* correspond to the elements of a 2-D coefficient array *c* of shape (xdeg + 1, ydeg + 1) in the order

$$c_{00}, c_{01}, c_{02}..., c_{10}, c_{11}, c_{12}...$$

and `np.dot(V, c.flat)` and `hermval2d(x, y, c)` will be the same up to roundoff. This equivalence is useful both for least squares fitting and for the evaluation of a large number of 2-D Hermite series of the same degrees and sample points.

**Parameters**

**x, y**

[array_like] Arrays of point coordinates, all of the same shape. The dtypes will be converted to either float64 or complex128 depending on whether any of the elements are complex. Scalars are converted to 1-D arrays.

**deg**

[list of ints] List of maximum degrees of the form [x_deg, y_deg].

**Returns**

**vander2d**

[ndarray] The shape of the returned matrix is `x.shape + (order,)`, where $order = (deg[0] + 1) * (deg[1] + 1)$. The dtype will be the same as the converted *x* and *y*.

**See also:**

*`hermvander`*, *`hermvander3d`*, *`hermval2d`*, *`hermval3d`*

### Notes

New in version 1.7.0.

`numpy.polynomial.hermite.`**`hermvander3d`**`(`*x*, *y*, *z*, *deg*`)`

Pseudo-Vandermonde matrix of given degrees.

Returns the pseudo-Vandermonde matrix of degrees *deg* and sample points *(x, y, z)*. If *l, m, n* are the given degrees in *x, y, z*, then The pseudo-Vandermonde matrix is defined by

$$V[..., (m+1)(n+1)i + (n+1)j + k] = H_i(x) * H_j(y) * H_k(z),$$

where *0 <= i <= l*, *0 <= j <= m*, and *0 <= j <= n*. The leading indices of *V* index the points *(x, y, z)* and the last index encodes the degrees of the Hermite polynomials.

If `V = hermvander3d(x, y, z, [xdeg, ydeg, zdeg])`, then the columns of *V* correspond to the elements of a 3-D coefficient array *c* of shape (xdeg + 1, ydeg + 1, zdeg + 1) in the order

$$c_{000}, c_{001}, c_{002}, ..., c_{010}, c_{011}, c_{012}, ...$$

and `np.dot(V, c.flat)` and `hermval3d(x, y, z, c)` will be the same up to roundoff. This equivalence is useful both for least squares fitting and for the evaluation of a large number of 3-D Hermite series of the same degrees and sample points.

**Parameters**

**x, y, z**

[array_like] Arrays of point coordinates, all of the same shape. The dtypes will be converted to either float64 or complex128 depending on whether any of the elements are complex. Scalars are converted to 1-D arrays.

**deg**

[list of ints] List of maximum degrees of the form [x_deg, y_deg, z_deg].

**Returns**

**vander3d**

[ndarray] The shape of the returned matrix is `x.shape + (order,)`, where $order = (deg[0]+1)*(deg[1]+1)*(deg[2]+1)$. The dtype will be the same as the converted *x*, *y*, and *z*.

**See also:**

*`hermvander`*, *`hermvander3d`*, *`hermval2d`*, *`hermval3d`*

### Notes

New in version 1.7.0.

`numpy.polynomial.hermite.`**`hermgauss`**`(`*deg*`)`

Gauss-Hermite quadrature.

Computes the sample points and weights for Gauss-Hermite quadrature. These sample points and weights will correctly integrate polynomials of degree $2*deg-1$ or less over the interval $[-\inf, \inf]$ with the weight function $f(x) = \exp(-x^2)$.

**Parameters**

**deg**

[int] Number of sample points and weights. It must be >= 1.

**Returns**

**x**

[ndarray] 1-D ndarray containing the sample points.

**y**

[ndarray] 1-D ndarray containing the weights.

## Notes

New in version 1.7.0.

The results have only been tested up to degree 100, higher degrees may be problematic. The weights are determined by using the fact that

$$w_k = c/(H'_n(x_k) * H_{n-1}(x_k))$$

where $c$ is a constant independent of $k$ and $x_k$ is the k'th root of $H_n$, and then scaling the results to get the right value when integrating 1.

`numpy.polynomial.hermite.`**`hermweight`**(*x*)

Weight function of the Hermite polynomials.

The weight function is $\exp(-x^2)$ and the interval of integration is $[-\inf, \inf]$. the Hermite polynomials are orthogonal, but not normalized, with respect to this weight function.

**Parameters**

**x**

[array_like] Values at which the weight function will be computed.

**Returns**

**w**

[ndarray] The weight function at *x*.

## Notes

New in version 1.7.0.

`numpy.polynomial.hermite.`**`hermcompanion`**(*c*)

Return the scaled companion matrix of c.

The basis polynomials are scaled so that the companion matrix is symmetric when *c* is an Hermite basis polynomial. This provides better eigenvalue estimates than the unscaled case and for basis polynomials the eigenvalues are guaranteed to be real if `numpy.linalg.eigvalsh` is used to obtain them.

**Parameters**

**c**

[array_like] 1-D array of Hermite series coefficients ordered from low to high degree.

**Returns**

**mat**

[ndarray] Scaled companion matrix of dimensions (deg, deg).

### Notes

New in version 1.7.0.

`numpy.polynomial.hermite.`**`hermfit`**(*x*, *y*, *deg*, *rcond=None*, *full=False*, *w=None*)

Least squares fit of Hermite series to data.

Return the coefficients of a Hermite series of degree *deg* that is the least squares fit to the data values *y* given at points *x*. If *y* is 1-D the returned coefficients will also be 1-D. If *y* is 2-D multiple fits are done, one for each column of *y*, and the resulting coefficients are stored in the corresponding columns of a 2-D return. The fitted polynomial(s) are in the form

$$p(x) = c_0 + c_1 * H_1(x) + ... + c_n * H_n(x),$$

where *n* is *deg*.

**Parameters**

**x**

[array_like, shape (M,)] x-coordinates of the M sample points `(x[i], y[i])`.

**y**

[array_like, shape (M,) or (M, K)] y-coordinates of the sample points. Several data sets of sample points sharing the same x-coordinates can be fitted at once by passing in a 2D-array that contains one dataset per column.

**deg**

[int or 1-D array_like] Degree(s) of the fitting polynomials. If *deg* is a single integer all terms up to and including the *deg*'th term are included in the fit. For NumPy versions >= 1.11.0 a list of integers specifying the degrees of the terms to include may be used instead.

**rcond**

[float, optional] Relative condition number of the fit. Singular values smaller than this relative to the largest singular value will be ignored. The default value is len(x)*eps, where eps is the relative precision of the float type, about 2e-16 in most cases.

**full**

[bool, optional] Switch determining nature of return value. When it is False (the default) just the coefficients are returned, when True diagnostic information from the singular value decomposition is also returned.

**w**

[array_like, shape (*M*,), optional] Weights. If not None, the contribution of each point `(x[i],y[i])` to the fit is weighted by *w[i]*. Ideally the weights are chosen so that the errors of the products `w[i]*y[i]` all have the same variance. The default value is None.

**Returns**

**coef**

[ndarray, shape (M,) or (M, K)] Hermite coefficients ordered from low to high. If *y* was 2-D, the coefficients for the data in column k of *y* are in column *k*.

**[residuals, rank, singular_values, rcond]**

[list] These values are only returned if *full* = True

resid – sum of squared residuals of the least squares fit rank – the numerical rank of the scaled Vandermonde matrix sv – singular values of the scaled Vandermonde matrix rcond – value of *rcond*.

For more details, see *linalg.lstsq*.

**Warns**

**RankWarning**

The rank of the coefficient matrix in the least-squares fit is deficient. The warning is only raised if *full* = False. The warnings can be turned off by

```
>>> import warnings
>>> warnings.simplefilter('ignore', np.RankWarning)
```

**See also:**

`chebfit`, `legfit`, `lagfit`, `polyfit`, `hermefit`

***`hermval`***

Evaluates a Hermite series.

***`hermvander`***

Vandermonde matrix of Hermite series.

***`hermweight`***

Hermite weight function

**`linalg.lstsq`**

Computes a least-squares fit from the matrix.

**`scipy.interpolate.UnivariateSpline`**

Computes spline fits.

## Notes

The solution is the coefficients of the Hermite series *p* that minimizes the sum of the weighted squared errors

$$E = \sum_j w_j^2 * |y_j - p(x_j)|^2,$$

where the $w_j$ are the weights. This problem is solved by setting up the (typically) overdetermined matrix equation

$$V(x) * c = w * y,$$

where *V* is the weighted pseudo Vandermonde matrix of *x*, *c* are the coefficients to be solved for, *w* are the weights, *y* are the observed values. This equation is then solved using the singular value decomposition of *V*.

If some of the singular values of $V$ are so small that they are neglected, then a *RankWarning* will be issued. This means that the coefficient values may be poorly determined. Using a lower order fit will usually get rid of the warning. The *rcond* parameter can also be set to a value smaller than its default, but the resulting fit may be spurious and have large contributions from roundoff error.

Fits using Hermite series are probably most useful when the data can be approximated by `sqrt(w(x)) * p(x)`, where *w(x)* is the Hermite weight. In that case the weight `sqrt(w(x[i]))` should be used together with data values `y[i]/sqrt(w(x[i]))`. The weight function is available as `hermweight`.

### References

[1]

### Examples

```
>>> from numpy.polynomial.hermite import hermfit, hermval
>>> x = np.linspace(-10, 10)
>>> err = np.random.randn(len(x))/10
>>> y = hermval(x, [1, 2, 3]) + err
>>> hermfit(x, y, 2)
array([1.0218, 1.9986, 2.9999]) # may vary
```

`numpy.polynomial.hermite.`**`hermtrim`**`(`*c*, *tol=0*`)`

Remove "small" "trailing" coefficients from a polynomial.

"Small" means "small in absolute value" and is controlled by the parameter *tol*; "trailing" means highest order coefficient(s), e.g., in `[0, 1, 1, 0, 0]` (which represents `0 + x + x**2 + 0*x**3 + 0*x**4`) both the 3-rd and 4-th order coefficients would be "trimmed."

**Parameters**

**c**

[array_like] 1-d array of coefficients, ordered from lowest order to highest.

**tol**

[number, optional] Trailing (i.e., highest order) elements with absolute value less than or equal to *tol* (default value is zero) are removed.

**Returns**

**trimmed**

[ndarray] 1-d array with trailing zeros removed. If the resulting series would be empty, a series containing a single zero is returned.

**Raises**

**ValueError**

If $tol < 0$

**See also:**

`trimseq`

### Examples

```
>>> from numpy.polynomial import polyutils as pu
>>> pu.trimcoef((0,0,3,0,5,0,0))
array([0.,  0.,  3.,  0.,  5.])
>>> pu.trimcoef((0,0,1e-3,0,1e-5,0,0),1e-3) # item == tol is trimmed
array([0.])
>>> i = complex(0,1) # works for complex
>>> pu.trimcoef((3e-4,1e-3*(1-i),5e-4,2e-5*(1+i)), 1e-3)
array([0.0003+0.j   , 0.001 -0.001j])
```

`numpy.polynomial.hermite.`**`hermline`**`(`*off*, *scl*`)`

Hermite series whose graph is a straight line.

**Parameters**

**off, scl**

[scalars] The specified line is given by `off + scl*x`.

**Returns**

**y**

[ndarray] This module's representation of the Hermite series for `off + scl*x`.

**See also:**

`polyline`, `chebline`

### Examples

```
>>> from numpy.polynomial.hermite import hermline, hermval
>>> hermval(0,hermline(3, 2))
3.0
>>> hermval(1,hermline(3, 2))
5.0
```

`numpy.polynomial.hermite.`**`herm2poly`**`(`*c*`)`

Convert a Hermite series to a polynomial.

Convert an array representing the coefficients of a Hermite series, ordered from lowest degree to highest, to an array of the coefficients of the equivalent polynomial (relative to the "standard" basis) ordered from lowest to highest degree.

**Parameters**

**c**

[array_like] 1-D array containing the Hermite series coefficients, ordered from lowest order term to highest.

**Returns**

**pol**

[ndarray] 1-D array containing the coefficients of the equivalent polynomial (relative to the "standard" basis) ordered from lowest order term to highest.

**See also:**

*poly2herm*

## Notes

The easy way to do conversions between polynomial basis sets is to use the convert method of a class instance.

## Examples

```
>>> from numpy.polynomial.hermite import herm2poly
>>> herm2poly([ 1.   ,  2.75 ,  0.5  ,  0.375])
array([0., 1., 2., 3.])
```

`numpy.polynomial.hermite.`**`poly2herm`**`(`*pol*`)`

Convert a polynomial to a Hermite series.

Convert an array representing the coefficients of a polynomial (relative to the "standard" basis) ordered from lowest degree to highest, to an array of the coefficients of the equivalent Hermite series, ordered from lowest to highest degree.

**Parameters**

**pol**

[array_like] 1-D array containing the polynomial coefficients

**Returns**

**c**

[ndarray] 1-D array containing the coefficients of the equivalent Hermite series.

**See also:**

*herm2poly*

## Notes

The easy way to do conversions between polynomial basis sets is to use the convert method of a class instance.

## Examples

```
>>> from numpy.polynomial.hermite import poly2herm
>>> poly2herm(np.arange(4))
array([1.   ,  2.75 ,  0.5  ,  0.375])
```

## See also

`numpy.polynomial`

New in version 1.6.0.

### HermiteE Series, “Probabilists” (`numpy.polynomial.hermite_e`)

This module provides a number of objects (mostly functions) useful for dealing with Hermite_e series, including a `HermiteE` class that encapsulates the usual arithmetic operations. (General information on how this module represents and works with such polynomials is in the docstring for its “parent” sub-package, `numpy.polynomial`).

## Classes

| | |
|---|---|
| `HermiteE`(coef[, domain, window]) | An HermiteE series class. |

**class** `numpy.polynomial.hermite_e.`**`HermiteE`**(*coef*, *domain=None*, *window=None*)

An HermiteE series class.

The HermiteE class provides the standard Python numerical methods ‘+’, ‘-‘, ‘*’, ‘//’, ‘%’, ‘divmod’, ‘**’, and ‘()’ as well as the attributes and methods listed in the `ABCPolyBase` documentation.

**Parameters**

**coef**

[array_like] HermiteE coefficients in order of increasing degree, i.e, `(1, 2, 3)` gives `1*He_0(x) + 2*He_1(X) + 3*He_2(x)`.

**domain**

[(2,) array_like, optional] Domain to use. The interval `[domain[0], domain[1]]` is mapped to the interval `[window[0], window[1]]` by shifting and scaling. The default value is [-1, 1].

**window**

[(2,) array_like, optional] Window, see `domain` for its use. The default value is [-1, 1].

New in version 1.6.0.

## Methods

| | |
|---|---|
| `__call__`(self, arg) | Call self as a function. |
| `basis`(deg[, domain, window]) | Series basis polynomial of degree *deg*. |
| `cast`(series[, domain, window]) | Convert series to series of this class. |
| `convert`(self[, domain, kind, window]) | Convert series to a different kind and/or domain and/or window. |
| `copy`(self) | Return a copy. |
| `cutdeg`(self, deg) | Truncate series to the given degree. |
| `degree`(self) | The degree of the series. |
| `deriv`(self[, m]) | Differentiate. |
| `fit`(x, y, deg[, domain, rcond, full, w, window]) | Least squares fit to data. |
| `fromroots`(roots[, domain, window]) | Return series instance that has the specified roots. |
| `has_samecoef`(self, other) | Check if coefficients match. |

Continued on next page

Table 116 – continued from previous page

| | |
|---|---|
| `has_samedomain`(self, other) | Check if domains match. |
| `has_sametype`(self, other) | Check if types match. |
| `has_samewindow`(self, other) | Check if windows match. |
| `identity`([domain, window]) | Identity function. |
| `integ`(self[, m, k, lbnd]) | Integrate. |
| `linspace`(self[, n, domain]) | Return x, y values at equally spaced points in domain. |
| `mapparms`(self) | Return the mapping parameters. |
| `roots`(self) | Return the roots of the series polynomial. |
| `trim`(self[, tol]) | Remove trailing coefficients |
| `truncate`(self, size) | Truncate series to length *size*. |

method

`HermiteE.`**`__call__`**(*self*, *arg*)

Call self as a function.

method

**`classmethod`** `HermiteE.`**`basis`**(*deg*, *domain=None*, *window=None*)

Series basis polynomial of degree *deg*.

Returns the series representing the basis polynomial of degree *deg*.

New in version 1.7.0.

**Parameters**

**deg**

[int] Degree of the basis polynomial for the series. Must be >= 0.

**domain**

[{None, array_like}, optional] If given, the array must be of the form `[beg, end]`, where `beg` and `end` are the endpoints of the domain. If None is given then the class domain is used. The default is None.

**window**

[{None, array_like}, optional] If given, the resulting array must be if the form `[beg, end]`, where `beg` and `end` are the endpoints of the window. If None is given then the class window is used. The default is None.

**Returns**

**new_series**

[series] A series with the coefficient of the *deg* term set to one and all others zero.

method

**`classmethod`** `HermiteE.`**`cast`**(*series*, *domain=None*, *window=None*)

Convert series to series of this class.

The *series* is expected to be an instance of some polynomial series of one of the types supported by by the numpy.polynomial module, but could be some other class that supports the convert method.

New in version 1.7.0.

**Parameters**

**series**

[series] The series instance to be converted.

**domain**

[{None, array_like}, optional] If given, the array must be of the form `[beg, end]`, where `beg` and `end` are the endpoints of the domain. If None is given then the class domain is used. The default is None.

**window**

[{None, array_like}, optional] If given, the resulting array must be if the form `[beg, end]`, where `beg` and `end` are the endpoints of the window. If None is given then the class window is used. The default is None.

**Returns**

**new_series**

[series] A series of the same kind as the calling class and equal to *series* when evaluated.

**See also:**

**`convert`**

similar instance method

method

`HermiteE.`**`convert`**(*self*, *domain=None*, *kind=None*, *window=None*)

Convert series to a different kind and/or domain and/or window.

**Parameters**

**domain**

[array_like, optional] The domain of the converted series. If the value is None, the default domain of *kind* is used.

**kind**

[class, optional] The polynomial series type class to which the current instance should be converted. If kind is None, then the class of the current instance is used.

**window**

[array_like, optional] The window of the converted series. If the value is None, the default window of *kind* is used.

**Returns**

**new_series**

[series] The returned class can be of different type than the current instance and/or have a different domain and/or different window.

### Notes

Conversion between domains and class types can result in numerically ill defined series.

method

`HermiteE.`**`copy`**`(`*self*`)`

Return a copy.

**Returns**

**new_series**

[series] Copy of self.

method

`HermiteE.`**`cutdeg`**`(`*self*, *deg*`)`

Truncate series to the given degree.

Reduce the degree of the series to *deg* by discarding the high order terms. If *deg* is greater than the current degree a copy of the current series is returned. This can be useful in least squares where the coefficients of the high degree terms may be very small.

New in version 1.5.0.

**Parameters**

**deg**

[non-negative int] The series is reduced to degree *deg* by discarding the high order terms. The value of *deg* must be a non-negative integer.

**Returns**

**new_series**

[series] New instance of series with reduced degree.

method

`HermiteE.`**`degree`**`(`*self*`)`

The degree of the series.

New in version 1.5.0.

**Returns**

**degree**

[int] Degree of the series, one less than the number of coefficients.

method

`HermiteE.`**`deriv`**`(`*self*, *m=1*`)`

Differentiate.

Return a series instance of that is the derivative of the current series.

**Parameters**

**m**

[non-negative int] Find the derivative of order *m*.

**Returns**

**new_series**

[series] A new series representing the derivative. The domain is the same as the domain of the differentiated series.

method

**classmethod** `HermiteE.`**`fit`**(*x*, *y*, *deg*, *domain=None*, *rcond=None*, *full=False*, *w=None*, *window=None*)

Least squares fit to data.

Return a series instance that is the least squares fit to the data *y* sampled at *x*. The domain of the returned instance can be specified and this will often result in a superior fit with less chance of ill conditioning.

**Parameters**

**x**

[array_like, shape (M,)] x-coordinates of the M sample points `(x[i], y[i])`.

**y**

[array_like, shape (M,) or (M, K)] y-coordinates of the sample points. Several data sets of sample points sharing the same x-coordinates can be fitted at once by passing in a 2D-array that contains one dataset per column.

**deg**

[int or 1-D array_like] Degree(s) of the fitting polynomials. If *deg* is a single integer all terms up to and including the *deg*'th term are included in the fit. For NumPy versions >= 1.11.0 a list of integers specifying the degrees of the terms to include may be used instead.

**domain**

[{None, [beg, end], []}, optional] Domain to use for the returned series. If `None`, then a minimal domain that covers the points *x* is chosen. If `[]` the class domain is used. The default value was the class domain in NumPy 1.4 and `None` in later versions. The `[]` option was added in numpy 1.5.0.

**rcond**

[float, optional] Relative condition number of the fit. Singular values smaller than this relative to the largest singular value will be ignored. The default value is len(x)*eps, where eps is the relative precision of the float type, about 2e-16 in most cases.

**full**

[bool, optional] Switch determining nature of return value. When it is False (the default) just the coefficients are returned, when True diagnostic information from the singular value decomposition is also returned.

**w**

[array_like, shape (M,), optional] Weights. If not None the contribution of each point `(x[i],y[i])` to the fit is weighted by *w[i]*. Ideally the weights are chosen so that the errors of the products `w[i]*y[i]` all have the same variance. The default value is None.

New in version 1.5.0.

**window**

[{[beg, end]}, optional] Window to use for the returned series. The default value is the default class domain

New in version 1.6.0.

**Returns**

**new_series**

[series] A series that represents the least squares fit to the data and has the domain and window specified in the call. If the coefficients for the unscaled and unshifted basis polynomials are of interest, do `new_series.convert().coef`.

**[resid, rank, sv, rcond]**

[list] These values are only returned if *full* = True

resid – sum of squared residuals of the least squares fit rank – the numerical rank of the scaled Vandermonde matrix sv – singular values of the scaled Vandermonde matrix rcond – value of *rcond*.

For more details, see *linalg.lstsq*.

method

**classmethod** `HermiteE.`**`fromroots`**(*roots*, *domain=[]*, *window=None*)

Return series instance that has the specified roots.

Returns a series representing the product `(x - r[0])*(x - r[1])*...*(x - r[n-1])`, where `r` is a list of roots.

**Parameters**

**roots**

[array_like] List of roots.

**domain**

[{[], None, array_like}, optional] Domain for the resulting series. If None the domain is the interval from the smallest root to the largest. If [] the domain is the class domain. The default is [].

**window**

[{None, array_like}, optional] Window for the returned series. If None the class window is used. The default is None.

**Returns**

**new_series**

[series] Series with the specified roots.

method

`HermiteE.`**`has_samecoef`**(*self*, *other*)

Check if coefficients match.

New in version 1.6.0.

**Parameters**

**other**

[class instance] The other class must have the `coef` attribute.

**Returns**

**bool**

[boolean] True if the coefficients are the same, False otherwise.

method

`HermiteE.`**`has_samedomain`**(*self*, *other*)

Check if domains match.

New in version 1.6.0.

**Parameters**

**other**

[class instance] The other class must have the `domain` attribute.

**Returns**

**bool**

[boolean] True if the domains are the same, False otherwise.

method

`HermiteE.`**`has_sametype`**(*self*, *other*)

Check if types match.

New in version 1.7.0.

**Parameters**

**other**

[object] Class instance.

**Returns**

**bool**

[boolean] True if other is same class as self

method

`HermiteE.`**`has_samewindow`**(*self*, *other*)

Check if windows match.

New in version 1.6.0.

**Parameters**

**other**

[class instance] The other class must have the `window` attribute.

**Returns**

**bool**

[boolean] True if the windows are the same, False otherwise.

method

**classmethod** `HermiteE.`**`identity`**(*domain=None*, *window=None*)

Identity function.

If `p` is the returned series, then `p(x) == x` for all values of x.

**Parameters**

**domain**

[{None, array_like}, optional] If given, the array must be of the form `[beg, end]`, where `beg` and `end` are the endpoints of the domain. If None is given then the class domain is used. The default is None.

**window**

[{None, array_like}, optional] If given, the resulting array must be if the form `[beg, end]`, where `beg` and `end` are the endpoints of the window. If None is given then the class window is used. The default is None.

**Returns**

**new_series**

[series] Series of representing the identity.

method

`HermiteE.`**`integ`**(*self*, *m=1*, *k=[]*, *lbnd=None*)

Integrate.

Return a series instance that is the definite integral of the current series.

**Parameters**

**m**

[non-negative int] The number of integrations to perform.

**k**

[array_like] Integration constants. The first constant is applied to the first integration, the second to the second, and so on. The list of values must less than or equal to *m* in length and any missing values are set to zero.

**lbnd**

[Scalar] The lower bound of the definite integral.

**Returns**

**new_series**

[series] A new series representing the integral. The domain is the same as the domain of the integrated series.

method

`HermiteE.`**`linspace`**(*self*, *n=100*, *domain=None*)

Return x, y values at equally spaced points in domain.

Returns the x, y values at *n* linearly spaced points across the domain. Here y is the value of the polynomial at the points x. By default the domain is the same as that of the series instance. This method is intended mostly as a plotting aid.

New in version 1.5.0.

**Parameters**

**n**

[int, optional] Number of point pairs to return. The default value is 100.

**domain**

[{None, array_like}, optional] If not None, the specified domain is used instead of that of the calling instance. It should be of the form `[beg,end]`. The default is None which case the class domain is used.

**Returns**

**x, y**

[ndarray] x is equal to linspace(self.domain[0], self.domain[1], n) and y is the series evaluated at element of `x`.

method

`HermiteE.`**`mapparms`**(*self*)

Return the mapping parameters.

The returned values define a linear map `off + scl*x` that is applied to the input arguments before the series is evaluated. The map depends on the `domain` and `window`; if the current `domain` is equal to the `window` the resulting map is the identity. If the coefficients of the series instance are to be used by themselves outside this class, then the linear function must be substituted for the `x` in the standard representation of the base polynomials.

**Returns**

**off, scl**

[float or complex] The mapping function is defined by `off + scl*x`.

### Notes

If the current domain is the interval `[l1, r1]` and the window is `[l2, r2]`, then the linear mapping function `L` is defined by the equations:

```
L(l1) = l2
L(r1) = r2
```

method

`HermiteE.`**`roots`**(*self*)

Return the roots of the series polynomial.

Compute the roots for the series. Note that the accuracy of the roots decrease the further outside the domain they lie.

**Returns**

**roots**

[ndarray] Array containing the roots of the series.

method

`HermiteE.`**`trim`**`(`*self*, *tol=0*`)`

Remove trailing coefficients

Remove trailing coefficients until a coefficient is reached whose absolute value greater than *tol* or the beginning of the series is reached. If all the coefficients would be removed the series is set to `[0]`. A new series instance is returned with the new coefficients. The current instance remains unchanged.

**Parameters**

**tol**

[non-negative number.] All trailing coefficients less than *tol* will be removed.

**Returns**

**new_series**

[series] Contains the new set of coefficients.

method

`HermiteE.`**`truncate`**`(`*self*, *size*`)`

Truncate series to length *size*.

Reduce the series to length *size* by discarding the high degree terms. The value of *size* must be a positive integer. This can be useful in least squares where the coefficients of the high degree terms may be very small.

**Parameters**

**size**

[positive int] The series is reduced to length *size* by discarding the high degree terms. The value of *size* must be a positive integer.

**Returns**

**new_series**

[series] New instance of series with truncated coefficients.

## Constants

| |
|---|
| *hermedomain* |
| *hermezero* |
| *hermeone* |
| *hermex* |

`numpy.polynomial.hermite_e.`**`hermedomain = array([-1, 1])`**

`numpy.polynomial.hermite_e.`**`hermezero = array([0])`**

`numpy.polynomial.hermite_e.`**`hermeone = array([1])`**

`numpy.polynomial.hermite_e.`**`hermex = array([0, 1])`**

## Arithmetic

| | |
|---|---|
| *hermeadd*(c1, c2) | Add one Hermite series to another. |
| *hermesub*(c1, c2) | Subtract one Hermite series from another. |
| *hermemulx*(c) | Multiply a Hermite series by x. |
| *hermemul*(c1, c2) | Multiply one Hermite series by another. |
| *hermediv*(c1, c2) | Divide one Hermite series by another. |
| *hermepow*(c, pow[, maxpower]) | Raise a Hermite series to a power. |
| *hermeval*(x, c[, tensor]) | Evaluate an HermiteE series at points x. |
| *hermeval2d*(x, y, c) | Evaluate a 2-D HermiteE series at points (x, y). |
| *hermeval3d*(x, y, z, c) | Evaluate a 3-D Hermite_e series at points (x, y, z). |
| *hermegrid2d*(x, y, c) | Evaluate a 2-D HermiteE series on the Cartesian product of x and y. |
| *hermegrid3d*(x, y, z, c) | Evaluate a 3-D HermiteE series on the Cartesian product of x, y, and z. |

`numpy.polynomial.hermite_e.`**`hermeadd`**`(`*c1*, *c2*`)`

Add one Hermite series to another.

Returns the sum of two Hermite series *c1* + *c2*. The arguments are sequences of coefficients ordered from lowest order term to highest, i.e., [1,2,3] represents the series `P_0 + 2*P_1 + 3*P_2`.

**Parameters**

**c1, c2**

[array_like] 1-D arrays of Hermite series coefficients ordered from low to high.

**Returns**

**out**

[ndarray] Array representing the Hermite series of their sum.

**See also:**

*hermesub*, *hermemulx*, *hermemul*, *hermediv*, *hermepow*

### Notes

Unlike multiplication, division, etc., the sum of two Hermite series is a Hermite series (without having to "reproject" the result onto the basis set) so addition, just like that of "standard" polynomials, is simply "component-wise."

### Examples

```
>>> from numpy.polynomial.hermite_e import hermeadd
>>> hermeadd([1, 2, 3], [1, 2, 3, 4])
array([2.,  4.,  6.,  4.])
```

`numpy.polynomial.hermite_e.`**`hermesub`**`(`*c1*, *c2*`)`

Subtract one Hermite series from another.

Returns the difference of two Hermite series *c1* - *c2*. The sequences of coefficients are from lowest order term to highest, i.e., [1,2,3] represents the series `P_0 + 2*P_1 + 3*P_2`.

**Parameters**

**c1, c2**

[array_like] 1-D arrays of Hermite series coefficients ordered from low to high.

**Returns**

**out**

[ndarray] Of Hermite series coefficients representing their difference.

**See also:**

*hermeadd*, *hermemulx*, *hermemul*, *hermediv*, *hermepow*

### Notes

Unlike multiplication, division, etc., the difference of two Hermite series is a Hermite series (without having to "reproject" the result onto the basis set) so subtraction, just like that of "standard" polynomials, is simply "component-wise."

### Examples

```
>>> from numpy.polynomial.hermite_e import hermesub
>>> hermesub([1, 2, 3, 4], [1, 2, 3])
array([0., 0., 0., 4.])
```

`numpy.polynomial.hermite_e.`**`hermemulx`**`(`*c*`)`

Multiply a Hermite series by x.

Multiply the Hermite series *c* by x, where x is the independent variable.

**Parameters**

**c**

[array_like] 1-D array of Hermite series coefficients ordered from low to high.

**Returns**

**out**

[ndarray] Array representing the result of the multiplication.

## Notes

The multiplication uses the recursion relationship for Hermite polynomials in the form

xP_i(x) = (P_{i + 1}(x) + iP_{i - 1}(x)))

## Examples

```
>>> from numpy.polynomial.hermite_e import hermemulx
>>> hermemulx([1, 2, 3])
array([2.,  7.,  2.,  3.])
```

`numpy.polynomial.hermite_e.`**`hermemul`**`(`*c1*, *c2*`)`

Multiply one Hermite series by another.

Returns the product of two Hermite series *c1* * *c2*. The arguments are sequences of coefficients, from lowest order "term" to highest, e.g., [1,2,3] represents the series `P_0 + 2*P_1 + 3*P_2`.

**Parameters**

**c1, c2**

[array_like] 1-D arrays of Hermite series coefficients ordered from low to high.

**Returns**

**out**

[ndarray] Of Hermite series coefficients representing their product.

**See also:**

*`hermeadd`*, *`hermesub`*, *`hermemulx`*, *`hermediv`*, *`hermepow`*

## Notes

In general, the (polynomial) product of two C-series results in terms that are not in the Hermite polynomial basis set. Thus, to express the product as a Hermite series, it is necessary to "reproject" the product onto said basis set, which may produce "unintuitive" (but correct) results; see Examples section below.

### Examples

```
>>> from numpy.polynomial.hermite_e import hermemul
>>> hermemul([1, 2, 3], [0, 1, 2])
array([14.,  15.,  28.,   7.,   6.])
```

`numpy.polynomial.hermite_e.`**`hermediv`**`(`*c1*, *c2*`)`

Divide one Hermite series by another.

Returns the quotient-with-remainder of two Hermite series *c1* / *c2*. The arguments are sequences of coefficients from lowest order "term" to highest, e.g., [1,2,3] represents the series `P_0 + 2*P_1 + 3*P_2`.

**Parameters**

**c1, c2**

[array_like] 1-D arrays of Hermite series coefficients ordered from low to high.

**Returns**

**[quo, rem]**

[ndarrays] Of Hermite series coefficients representing the quotient and remainder.

**See also:**

*`hermeadd`*, *`hermesub`*, *`hermemulx`*, *`hermemul`*, *`hermepow`*

### Notes

In general, the (polynomial) division of one Hermite series by another results in quotient and remainder terms that are not in the Hermite polynomial basis set. Thus, to express these results as a Hermite series, it is necessary to "reproject" the results onto the Hermite basis set, which may produce "unintuitive" (but correct) results; see Examples section below.

### Examples

```
>>> from numpy.polynomial.hermite_e import hermediv
>>> hermediv([ 14.,  15.,  28.,   7.,   6.], [0, 1, 2])
(array([1., 2., 3.]), array([0.]))
>>> hermediv([ 15.,  17.,  28.,   7.,   6.], [0, 1, 2])
(array([1., 2., 3.]), array([1., 2.]))
```

`numpy.polynomial.hermite_e.`**`hermepow`**`(`*c*, *pow*, *maxpower=16*`)`

Raise a Hermite series to a power.

Returns the Hermite series *c* raised to the power *pow*. The argument *c* is a sequence of coefficients ordered from low to high. i.e., [1,2,3] is the series `P_0 + 2*P_1 + 3*P_2.`

**Parameters**

**c**

[array_like] 1-D array of Hermite series coefficients ordered from low to high.

**pow**

[integer] Power to which the series will be raised

**maxpower**

[integer, optional] Maximum power allowed. This is mainly to limit growth of the series to unmanageable size. Default is 16

**Returns**

**coef**

[ndarray] Hermite series of power.

**See also:**

`hermeadd`, `hermesub`, `hermemulx`, `hermemul`, `hermediv`

## Examples

```
>>> from numpy.polynomial.hermite_e import hermepow
>>> hermepow([1, 2, 3], 2)
array([23.,  28.,  46.,  12.,   9.])
```

`numpy.polynomial.hermite_e.`**`hermeval`**(*x*, *c*, *tensor=True*)

Evaluate an HermiteE series at points x.

If *c* is of length *n* + *1*, this function returns the value:

$$p(x) = c_0 * He_0(x) + c_1 * He_1(x) + ... + c_n * He_n(x)$$

The parameter *x* is converted to an array only if it is a tuple or a list, otherwise it is treated as a scalar. In either case, either *x* or its elements must support multiplication and addition both with themselves and with the elements of *c*.

If *c* is a 1-D array, then *p(x)* will have the same shape as *x*. If *c* is multidimensional, then the shape of the result depends on the value of *tensor*. If *tensor* is true the shape will be c.shape[1:] + x.shape. If *tensor* is false the shape will be c.shape[1:]. Note that scalars have shape (,).

Trailing zeros in the coefficients will be used in the evaluation, so they should be avoided if efficiency is a concern.

**Parameters**

**x**

[array_like, compatible object] If *x* is a list or tuple, it is converted to an ndarray, otherwise it is left unchanged and treated as a scalar. In either case, *x* or its elements must support addition and multiplication with with themselves and with the elements of *c*.

**c**

[array_like] Array of coefficients ordered so that the coefficients for terms of degree n are contained in c[n]. If *c* is multidimensional the remaining indices enumerate multiple polynomials. In the two dimensional case the coefficients may be thought of as stored in the columns of *c*.

**tensor**

[boolean, optional] If True, the shape of the coefficient array is extended with ones on the right, one for each dimension of $x$. Scalars have dimension 0 for this action. The result is that every column of coefficients in $c$ is evaluated for every element of $x$. If False, $x$ is broadcast over the columns of $c$ for the evaluation. This keyword is useful when $c$ is multidimensional. The default value is True.

New in version 1.7.0.

**Returns**

**values**

[ndarray, algebra_like] The shape of the return value is described above.

**See also:**

*hermeval2d*, *hermegrid2d*, *hermeval3d*, *hermegrid3d*

## Notes

The evaluation uses Clenshaw recursion, aka synthetic division.

## Examples

```
>>> from numpy.polynomial.hermite_e import hermeval
>>> coef = [1,2,3]
>>> hermeval(1, coef)
3.0
>>> hermeval([[1,2],[3,4]], coef)
array([[ 3., 14.],
       [31., 54.]])
```

`numpy.polynomial.hermite_e.`**`hermeval2d`**(*x*, *y*, *c*)

Evaluate a 2-D HermiteE series at points (x, y).

This function returns the values:

$$p(x,y) = \sum_{i,j} c_{i,j} * He_i(x) * He_j(y)$$

The parameters $x$ and $y$ are converted to arrays only if they are tuples or a lists, otherwise they are treated as a scalars and they must have the same shape after conversion. In either case, either $x$ and $y$ or their elements must support multiplication and addition both with themselves and with the elements of $c$.

If $c$ is a 1-D array a one is implicitly appended to its shape to make it 2-D. The shape of the result will be c.shape[2:] + x.shape.

**Parameters**

**x, y**

[array_like, compatible objects] The two dimensional series is evaluated at the points *(x, y)*, where $x$ and $y$ must have the same shape. If $x$ or $y$ is a list or tuple, it is first converted to an ndarray, otherwise it is left unchanged and if it isn't an ndarray it is treated as a scalar.

**c**

[array_like] Array of coefficients ordered so that the coefficient of the term of multi-degree i,j is contained in `c[i,j]`. If *c* has dimension greater than two the remaining indices enumerate multiple sets of coefficients.

**Returns**

**values**

[ndarray, compatible object] The values of the two dimensional polynomial at points formed with pairs of corresponding values from *x* and *y*.

**See also:**

*hermeval*, *hermegrid2d*, *hermeval3d*, *hermegrid3d*

## Notes

New in version 1.7.0.

`numpy.polynomial.hermite_e.`**`hermeval3d`**`(x, y, z, c)`

Evaluate a 3-D Hermite_e series at points (x, y, z).

This function returns the values:

$$p(x,y,z) = \sum_{i,j,k} c_{i,j,k} * He_i(x) * He_j(y) * He_k(z)$$

The parameters *x*, *y*, and *z* are converted to arrays only if they are tuples or a lists, otherwise they are treated as a scalars and they must have the same shape after conversion. In either case, either *x*, *y*, and *z* or their elements must support multiplication and addition both with themselves and with the elements of *c*.

If *c* has fewer than 3 dimensions, ones are implicitly appended to its shape to make it 3-D. The shape of the result will be c.shape[3:] + x.shape.

**Parameters**

**x, y, z**

[array_like, compatible object] The three dimensional series is evaluated at the points *(x, y, z)*, where *x*, *y*, and *z* must have the same shape. If any of *x*, *y*, or *z* is a list or tuple, it is first converted to an ndarray, otherwise it is left unchanged and if it isn't an ndarray it is treated as a scalar.

**c**

[array_like] Array of coefficients ordered so that the coefficient of the term of multi-degree i,j,k is contained in `c[i,j,k]`. If *c* has dimension greater than 3 the remaining indices enumerate multiple sets of coefficients.

**Returns**

**values**

[ndarray, compatible object] The values of the multidimensional polynomial on points formed with triples of corresponding values from *x*, *y*, and *z*.

**See also:**

*hermeval*, *hermeval2d*, *hermegrid2d*, *hermegrid3d*

## Notes

New in version 1.7.0.

`numpy.polynomial.hermite_e.`**`hermegrid2d`**(*x*, *y*, *c*)

Evaluate a 2-D HermiteE series on the Cartesian product of x and y.

This function returns the values:

$$p(a,b) = \sum_{i,j} c_{i,j} * H_i(a) * H_j(b)$$

where the points *(a, b)* consist of all pairs formed by taking *a* from *x* and *b* from *y*. The resulting points form a grid with *x* in the first dimension and *y* in the second.

The parameters *x* and *y* are converted to arrays only if they are tuples or a lists, otherwise they are treated as a scalars. In either case, either *x* and *y* or their elements must support multiplication and addition both with themselves and with the elements of *c*.

If *c* has fewer than two dimensions, ones are implicitly appended to its shape to make it 2-D. The shape of the result will be c.shape[2:] + x.shape.

**Parameters**

**x, y**

[array_like, compatible objects] The two dimensional series is evaluated at the points in the Cartesian product of *x* and *y*. If *x* or *y* is a list or tuple, it is first converted to an ndarray, otherwise it is left unchanged and, if it isn't an ndarray, it is treated as a scalar.

**c**

[array_like] Array of coefficients ordered so that the coefficients for terms of degree i,j are contained in `c[i,j]`. If *c* has dimension greater than two the remaining indices enumerate multiple sets of coefficients.

**Returns**

**values**

[ndarray, compatible object] The values of the two dimensional polynomial at points in the Cartesian product of *x* and *y*.

**See also:**

*`hermeval`*, *`hermeval2d`*, *`hermeval3d`*, *`hermegrid3d`*

## Notes

New in version 1.7.0.

`numpy.polynomial.hermite_e.`**`hermegrid3d`**(*x*, *y*, *z*, *c*)

Evaluate a 3-D HermiteE series on the Cartesian product of x, y, and z.

This function returns the values:

$$p(a,b,c) = \sum_{i,j,k} c_{i,j,k} * He_i(a) * He_j(b) * He_k(c)$$

where the points *(a, b, c)* consist of all triples formed by taking *a* from *x*, *b* from *y*, and *c* from *z*. The resulting points form a grid with *x* in the first dimension, *y* in the second, and *z* in the third.

The parameters *x*, *y*, and *z* are converted to arrays only if they are tuples or a lists, otherwise they are treated as a scalars. In either case, either *x*, *y*, and *z* or their elements must support multiplication and addition both with themselves and with the elements of *c*.

If *c* has fewer than three dimensions, ones are implicitly appended to its shape to make it 3-D. The shape of the result will be c.shape[3:] + x.shape + y.shape + z.shape.

**Parameters**

**x, y, z**

[array_like, compatible objects] The three dimensional series is evaluated at the points in the Cartesian product of *x*, *y*, and *z*. If *x*,'y', or *z* is a list or tuple, it is first converted to an ndarray, otherwise it is left unchanged and, if it isn't an ndarray, it is treated as a scalar.

**c**

[array_like] Array of coefficients ordered so that the coefficients for terms of degree i,j are contained in `c[i,j]`. If *c* has dimension greater than two the remaining indices enumerate multiple sets of coefficients.

**Returns**

**values**

[ndarray, compatible object] The values of the two dimensional polynomial at points in the Cartesian product of *x* and *y*.

**See also:**

`hermeval`, `hermeval2d`, `hermegrid2d`, `hermeval3d`

## Notes

New in version 1.7.0.

## Calculus

| | |
|---|---|
| `hermeder`(c[, m, scl, axis]) | Differentiate a Hermite_e series. |
| `hermeint`(c[, m, k, lbnd, scl, axis]) | Integrate a Hermite_e series. |

`numpy.polynomial.hermite_e.`**`hermeder`**(*c*, *m=1*, *scl=1*, *axis=0*)

Differentiate a Hermite_e series.

Returns the series coefficients *c* differentiated *m* times along *axis*. At each iteration the result is multiplied by *scl* (the scaling factor is for use in a linear change of variable). The argument *c* is an array of coefficients from low to high degree along each axis, e.g., [1,2,3] represents the series `1*He_0 + 2*He_1 + 3*He_2` while [[1,2],[1,2]] represents `1*He_0(x)*He_0(y) + 1*He_1(x)*He_0(y) + 2*He_0(x)*He_1(y) + 2*He_1(x)*He_1(y)` if axis=0 is `x` and axis=1 is `y`.

**Parameters**

**c**

[array_like] Array of Hermite_e series coefficients. If *c* is multidimensional the different axis correspond to different variables with the degree in each axis given by the corresponding index.

**m**

[int, optional] Number of derivatives taken, must be non-negative. (Default: 1)

**scl**

[scalar, optional] Each differentiation is multiplied by *scl*. The end result is multiplication by `scl**m`. This is for use in a linear change of variable. (Default: 1)

**axis**

[int, optional] Axis over which the derivative is taken. (Default: 0).

New in version 1.7.0.

**Returns**

**der**

[ndarray] Hermite series of the derivative.

**See also:**

`hermeint`

### Notes

In general, the result of differentiating a Hermite series does not resemble the same operation on a power series. Thus the result of this function may be "unintuitive," albeit correct; see Examples section below.

### Examples

```
>>> from numpy.polynomial.hermite_e import hermeder
>>> hermeder([ 1.,  1.,  1.,  1.])
array([1.,  2.,  3.])
>>> hermeder([-0.25,  1.,  1./2.,  1./3.,  1./4 ], m=2)
array([1.,  2.,  3.])
```

`numpy.polynomial.hermite_e.`**`hermeint`**(*c*, *m=1*, *k=[]*, *lbnd=0*, *scl=1*, *axis=0*)

Integrate a Hermite_e series.

Returns the Hermite_e series coefficients *c* integrated *m* times from *lbnd* along *axis*. At each iteration the resulting series is **multiplied** by *scl* and an integration constant, *k*, is added. The scaling factor is for use in a linear change of variable. ("Buyer beware": note that, depending on what one is doing, one may want *scl* to be the reciprocal of what one might expect; for more information, see the Notes section below.) The argument *c* is an array of coefficients from low to high degree along each axis, e.g., [1,2,3] represents the series `H_0 + 2*H_1 + 3*H_2` while [[1,2],[1,2]] represents `1*H_0(x)*H_0(y) + 1*H_1(x)*H_0(y) + 2*H_0(x)*H_1(y) + 2*H_1(x)*H_1(y)` if axis=0 is `x` and axis=1 is `y`.

**Parameters**

**c**

[array_like] Array of Hermite_e series coefficients. If c is multidimensional the different axis correspond to different variables with the degree in each axis given by the corresponding index.

**m**

[int, optional] Order of integration, must be positive. (Default: 1)

**k**

[{[], list, scalar}, optional] Integration constant(s). The value of the first integral at `lbnd` is the first value in the list, the value of the second integral at `lbnd` is the second value, etc. If `k == []` (the default), all constants are set to zero. If `m == 1`, a single scalar can be given instead of a list.

**lbnd**

[scalar, optional] The lower bound of the integral. (Default: 0)

**scl**

[scalar, optional] Following each integration the result is *multiplied* by *scl* before the integration constant is added. (Default: 1)

**axis**

[int, optional] Axis over which the integral is taken. (Default: 0).

New in version 1.7.0.

**Returns**

**S**

[ndarray] Hermite_e series coefficients of the integral.

**Raises**

**ValueError**

If `m < 0`, `len(k) > m`, `np.ndim(lbnd) != 0`, or `np.ndim(scl) != 0`.

**See also:**

*hermeder*

## Notes

Note that the result of each integration is *multiplied* by *scl*. Why is this important to note? Say one is making a linear change of variable $u = ax + b$ in an integral relative to $x$. Then $dx = du/a$, so one will need to set *scl* equal to $1/a$ - perhaps not what one would have first thought.

Also note that, in general, the result of integrating a C-series needs to be "reprojected" onto the C-series basis set. Thus, typically, the result of this function is "unintuitive," albeit correct; see Examples section below.

## Examples

```
>>> from numpy.polynomial.hermite_e import hermeint
>>> hermeint([1, 2, 3]) # integrate once, value 0 at 0.
array([1., 1., 1., 1.])
>>> hermeint([1, 2, 3], m=2) # integrate twice, value & deriv 0 at 0
array([-0.25      ,  1.        ,  0.5       ,  0.33333333,  0.25      ]) # may␣
↪vary
>>> hermeint([1, 2, 3], k=1) # integrate once, value 1 at 0.
array([2., 1., 1., 1.])
>>> hermeint([1, 2, 3], lbnd=-1) # integrate once, value 0 at -1
```

(continues on next page)

(continued from previous page)

```
array([-1.,  1.,  1.,  1.])
>>> hermeint([1, 2, 3], m=2, k=[1, 2], lbnd=-1)
array([ 1.83333333,  0.        ,  0.5       ,  0.33333333,  0.25      ]) # may␣
↪vary
```

## Misc Functions

| | |
|---|---|
| `hermefromroots`(roots) | Generate a HermiteE series with given roots. |
| `hermeroots`(c) | Compute the roots of a HermiteE series. |
| `hermevander`(x, deg) | Pseudo-Vandermonde matrix of given degree. |
| `hermevander2d`(x, y, deg) | Pseudo-Vandermonde matrix of given degrees. |
| `hermevander3d`(x, y, z, deg) | Pseudo-Vandermonde matrix of given degrees. |
| `hermegauss`(deg) | Gauss-HermiteE quadrature. |
| `hermeweight`(x) | Weight function of the Hermite_e polynomials. |
| `hermecompanion`(c) | Return the scaled companion matrix of c. |
| `hermefit`(x, y, deg[, rcond, full, w]) | Least squares fit of Hermite series to data. |
| `hermetrim`(c[, tol]) | Remove “small” “trailing” coefficients from a polynomial. |
| `hermeline`(off, scl) | Hermite series whose graph is a straight line. |
| `herme2poly`(c) | Convert a Hermite series to a polynomial. |
| `poly2herme`(pol) | Convert a polynomial to a Hermite series. |

`numpy.polynomial.hermite_e.`**`hermefromroots`**`(`*roots*`)`

Generate a HermiteE series with given roots.

The function returns the coefficients of the polynomial

$$p(x) = (x - r_0) * (x - r_1) * ... * (x - r_n),$$

in HermiteE form, where the *r_n* are the roots specified in *roots*. If a zero has multiplicity n, then it must appear in *roots* n times. For instance, if 2 is a root of multiplicity three and 3 is a root of multiplicity 2, then *roots* looks something like [2, 2, 2, 3, 3]. The roots can appear in any order.

If the returned coefficients are *c*, then

$$p(x) = c_0 + c_1 * He_1(x) + ... + c_n * He_n(x)$$

The coefficient of the last term is not generally 1 for monic polynomials in HermiteE form.

**Parameters**

**roots**

[array_like] Sequence containing the roots.

**Returns**

**out**

[ndarray] 1-D array of coefficients. If all roots are real then *out* is a real array, if some of the roots are complex, then *out* is complex even if all the coefficients in the result are real (see Examples below).

**See also:**

`polyfromroots`, `legfromroots`, `lagfromroots`, `hermfromroots`, `chebfromroots`

### Examples

```
>>> from numpy.polynomial.hermite_e import hermefromroots, hermeval
>>> coef = hermefromroots((-1, 0, 1))
>>> hermeval((-1, 0, 1), coef)
array([0., 0., 0.])
>>> coef = hermefromroots((-1j, 1j))
>>> hermeval((-1j, 1j), coef)
array([0.+0.j, 0.+0.j])
```

`numpy.polynomial.hermite_e.`**`hermeroots`**(*c*)

Compute the roots of a HermiteE series.

Return the roots (a.k.a. "zeros") of the polynomial

$$p(x) = \sum_i c[i] * He_i(x).$$

**Parameters**

**c**

[1-D array_like] 1-D array of coefficients.

**Returns**

**out**

[ndarray] Array of the roots of the series. If all the roots are real, then *out* is also real, otherwise it is complex.

**See also:**

`polyroots`, `legroots`, `lagroots`, `hermroots`, `chebroots`

### Notes

The root estimates are obtained as the eigenvalues of the companion matrix, Roots far from the origin of the complex plane may have large errors due to the numerical instability of the series for such values. Roots with multiplicity greater than 1 will also show larger errors as the value of the series near such points is relatively insensitive to errors in the roots. Isolated roots near the origin can be improved by a few iterations of Newton's method.

The HermiteE series basis polynomials aren't powers of *x* so the results of this function may seem unintuitive.

### Examples

```
>>> from numpy.polynomial.hermite_e import hermeroots, hermefromroots
>>> coef = hermefromroots([-1, 0, 1])
>>> coef
array([0., 2., 0., 1.])
>>> hermeroots(coef)
array([-1.,  0.,  1.]) # may vary
```

`numpy.polynomial.hermite_e.`**`hermevander`**(*x*, *deg*)

Pseudo-Vandermonde matrix of given degree.

Returns the pseudo-Vandermonde matrix of degree *deg* and sample points *x*. The pseudo-Vandermonde matrix is defined by

$$V[..., i] = He_i(x),$$

where *0 <= i <= deg*. The leading indices of *V* index the elements of *x* and the last index is the degree of the HermiteE polynomial.

If *c* is a 1-D array of coefficients of length *n + 1* and *V* is the array `V = hermevander(x, n)`, then `np.dot(V, c)` and `hermeval(x, c)` are the same up to roundoff. This equivalence is useful both for least squares fitting and for the evaluation of a large number of HermiteE series of the same degree and sample points.

**Parameters**

**x**

[array_like] Array of points. The dtype is converted to float64 or complex128 depending on whether any of the elements are complex. If *x* is scalar it is converted to a 1-D array.

**deg**

[int] Degree of the resulting matrix.

**Returns**

**vander**

[ndarray] The pseudo-Vandermonde matrix. The shape of the returned matrix is `x.shape + (deg + 1,)`, where The last index is the degree of the corresponding HermiteE polynomial. The dtype will be the same as the converted *x*.

## Examples

```
>>> from numpy.polynomial.hermite_e import hermevander
>>> x = np.array([-1, 0, 1])
>>> hermevander(x, 3)
array([[ 1., -1.,  0.,  2.],
       [ 1.,  0., -1., -0.],
       [ 1.,  1.,  0., -2.]])
```

`numpy.polynomial.hermite_e.`**`hermevander2d`**(*x, y, deg*)

Pseudo-Vandermonde matrix of given degrees.

Returns the pseudo-Vandermonde matrix of degrees *deg* and sample points *(x, y)*. The pseudo-Vandermonde matrix is defined by

$$V[..., (deg[1] + 1) * i + j] = He_i(x) * He_j(y),$$

where *0 <= i <= deg[0]* and *0 <= j <= deg[1]*. The leading indices of *V* index the points *(x, y)* and the last index encodes the degrees of the HermiteE polynomials.

If `V = hermevander2d(x, y, [xdeg, ydeg])`, then the columns of *V* correspond to the elements of a 2-D coefficient array *c* of shape (xdeg + 1, ydeg + 1) in the order

$$c_{00}, c_{01}, c_{02}..., c_{10}, c_{11}, c_{12}...$$

and `np.dot(V, c.flat)` and `hermeval2d(x, y, c)` will be the same up to roundoff. This equivalence is useful both for least squares fitting and for the evaluation of a large number of 2-D HermiteE series of the same degrees and sample points.

**Parameters**

**x, y**

[array_like] Arrays of point coordinates, all of the same shape. The dtypes will be converted to either float64 or complex128 depending on whether any of the elements are complex. Scalars are converted to 1-D arrays.

**deg**

[list of ints] List of maximum degrees of the form [x_deg, y_deg].

**Returns**

**vander2d**

[ndarray] The shape of the returned matrix is `x.shape + (order,)`, where $order = (deg[0] + 1) * (deg[1] + 1)$. The dtype will be the same as the converted *x* and *y*.

**See also:**

*hermevander*, *hermevander3d*, *hermeval2d*, *hermeval3d*

## Notes

New in version 1.7.0.

`numpy.polynomial.hermite_e.`**`hermevander3d`**`(`*x*, *y*, *z*, *deg*`)`

Pseudo-Vandermonde matrix of given degrees.

Returns the pseudo-Vandermonde matrix of degrees *deg* and sample points *(x, y, z)*. If *l, m, n* are the given degrees in *x, y, z*, then Hehe pseudo-Vandermonde matrix is defined by

$$V[..., (m+1)(n+1)i + (n+1)j + k] = He_i(x) * He_j(y) * He_k(z),$$

where *0 <= i <= l*, *0 <= j <= m*, and *0 <= j <= n*. The leading indices of $V$ index the points *(x, y, z)* and the last index encodes the degrees of the HermiteE polynomials.

If `V = hermevander3d(x, y, z, [xdeg, ydeg, zdeg])`, then the columns of $V$ correspond to the elements of a 3-D coefficient array *c* of shape (xdeg + 1, ydeg + 1, zdeg + 1) in the order

$$c_{000}, c_{001}, c_{002}, ..., c_{010}, c_{011}, c_{012}, ...$$

and `np.dot(V, c.flat)` and `hermeval3d(x, y, z, c)` will be the same up to roundoff. This equivalence is useful both for least squares fitting and for the evaluation of a large number of 3-D HermiteE series of the same degrees and sample points.

**Parameters**

**x, y, z**

[array_like] Arrays of point coordinates, all of the same shape. The dtypes will be converted to either float64 or complex128 depending on whether any of the elements are complex. Scalars are converted to 1-D arrays.

**deg**

[list of ints] List of maximum degrees of the form [x_deg, y_deg, z_deg].

**Returns**

**vander3d**

[ndarray] The shape of the returned matrix is `x.shape + (order,)`, where $order = (deg[0]+1)*(deg[1]+1)*(deg[2]+1)$. The dtype will be the same as the converted *x*, *y*, and *z*.

**See also:**

*hermevander*, *hermevander3d*, *hermeval2d*, *hermeval3d*

### Notes

New in version 1.7.0.

`numpy.polynomial.hermite_e.`**`hermegauss`**`(`*deg*`)`

Gauss-HermiteE quadrature.

Computes the sample points and weights for Gauss-HermiteE quadrature. These sample points and weights will correctly integrate polynomials of degree $2*deg-1$ or less over the interval $[-\inf, \inf]$ with the weight function $f(x) = \exp(-x^2/2)$.

**Parameters**

**deg**

[int] Number of sample points and weights. It must be >= 1.

**Returns**

**x**

[ndarray] 1-D ndarray containing the sample points.

**y**

[ndarray] 1-D ndarray containing the weights.

### Notes

New in version 1.7.0.

The results have only been tested up to degree 100, higher degrees may be problematic. The weights are determined by using the fact that

$$w_k = c/(He'_n(x_k) * He_{n-1}(x_k))$$

where $c$ is a constant independent of $k$ and $x_k$ is the k'th root of $He_n$, and then scaling the results to get the right value when integrating 1.

`numpy.polynomial.hermite_e.`**`hermeweight`**`(`*x*`)`

Weight function of the Hermite_e polynomials.

The weight function is $\exp(-x^2/2)$ and the interval of integration is $[-\inf, \inf]$. the HermiteE polynomials are orthogonal, but not normalized, with respect to this weight function.

**Parameters**

**x**

[array_like] Values at which the weight function will be computed.

**Returns**

**w**

[ndarray] The weight function at *x*.

## Notes

New in version 1.7.0.

`numpy.polynomial.hermite_e.`**`hermecompanion`**(*c*)

Return the scaled companion matrix of c.

The basis polynomials are scaled so that the companion matrix is symmetric when *c* is an HermiteE basis polynomial. This provides better eigenvalue estimates than the unscaled case and for basis polynomials the eigenvalues are guaranteed to be real if `numpy.linalg.eigvalsh` is used to obtain them.

**Parameters**

**c**

[array_like] 1-D array of HermiteE series coefficients ordered from low to high degree.

**Returns**

**mat**

[ndarray] Scaled companion matrix of dimensions (deg, deg).

## Notes

New in version 1.7.0.

`numpy.polynomial.hermite_e.`**`hermefit`**(*x*, *y*, *deg*, *rcond=None*, *full=False*, *w=None*)

Least squares fit of Hermite series to data.

Return the coefficients of a HermiteE series of degree *deg* that is the least squares fit to the data values *y* given at points *x*. If *y* is 1-D the returned coefficients will also be 1-D. If *y* is 2-D multiple fits are done, one for each column of *y*, and the resulting coefficients are stored in the corresponding columns of a 2-D return. The fitted polynomial(s) are in the form

$$p(x) = c_0 + c_1 * He_1(x) + ... + c_n * He_n(x),$$

where *n* is *deg*.

**Parameters**

**x**

[array_like, shape (M,)] x-coordinates of the M sample points `(x[i], y[i])`.

**y**

[array_like, shape (M,) or (M, K)] y-coordinates of the sample points. Several data sets of sample points sharing the same x-coordinates can be fitted at once by passing in a 2D-array that contains one dataset per column.

**deg**

[int or 1-D array_like] Degree(s) of the fitting polynomials. If *deg* is a single integer all terms up to and including the *deg*'th term are included in the fit. For NumPy versions >= 1.11.0 a list of integers specifying the degrees of the terms to include may be used instead.

**rcond**

[float, optional] Relative condition number of the fit. Singular values smaller than this relative to the largest singular value will be ignored. The default value is len(x)*eps, where eps is the relative precision of the float type, about 2e-16 in most cases.

**full**

[bool, optional] Switch determining nature of return value. When it is False (the default) just the coefficients are returned, when True diagnostic information from the singular value decomposition is also returned.

**w**

[array_like, shape (*M*,), optional] Weights. If not None, the contribution of each point `(x[i],y[i])` to the fit is weighted by *w[i]*. Ideally the weights are chosen so that the errors of the products `w[i]*y[i]` all have the same variance. The default value is None.

**Returns**

**coef**

[ndarray, shape (M,) or (M, K)] Hermite coefficients ordered from low to high. If *y* was 2-D, the coefficients for the data in column k of *y* are in column *k*.

**[residuals, rank, singular_values, rcond]**

[list] These values are only returned if *full* = True

resid – sum of squared residuals of the least squares fit rank – the numerical rank of the scaled Vandermonde matrix sv – singular values of the scaled Vandermonde matrix rcond – value of *rcond*.

For more details, see *linalg.lstsq*.

**Warns**

**RankWarning**

The rank of the coefficient matrix in the least-squares fit is deficient. The warning is only raised if *full* = False. The warnings can be turned off by

```
>>> import warnings
>>> warnings.simplefilter('ignore', np.RankWarning)
```

**See also:**

`chebfit`, `legfit`, `polyfit`, `hermfit`, `polyfit`

***hermeval***

Evaluates a Hermite series.

***hermevander***

pseudo Vandermonde matrix of Hermite series.

**`hermeweight`**

HermiteE weight function.

**`linalg.lstsq`**

Computes a least-squares fit from the matrix.

**`scipy.interpolate.UnivariateSpline`**

Computes spline fits.

### Notes

The solution is the coefficients of the HermiteE series *p* that minimizes the sum of the weighted squared errors

$$E = \sum_j w_j^2 * |y_j - p(x_j)|^2,$$

where the $w_j$ are the weights. This problem is solved by setting up the (typically) overdetermined matrix equation

$$V(x) * c = w * y,$$

where *V* is the pseudo Vandermonde matrix of *x*, the elements of *c* are the coefficients to be solved for, and the elements of *y* are the observed values. This equation is then solved using the singular value decomposition of *V*.

If some of the singular values of *V* are so small that they are neglected, then a *RankWarning* will be issued. This means that the coefficient values may be poorly determined. Using a lower order fit will usually get rid of the warning. The *rcond* parameter can also be set to a value smaller than its default, but the resulting fit may be spurious and have large contributions from roundoff error.

Fits using HermiteE series are probably most useful when the data can be approximated by `sqrt(w(x)) * p(x)`, where *w(x)* is the HermiteE weight. In that case the weight `sqrt(w(x[i]))` should be used together with data values `y[i]/sqrt(w(x[i]))`. The weight function is available as `hermeweight`.

### References

[1]

### Examples

```
>>> from numpy.polynomial.hermite_e import hermefit, hermeval
>>> x = np.linspace(-10, 10)
>>> np.random.seed(123)
>>> err = np.random.randn(len(x))/10
>>> y = hermeval(x, [1, 2, 3]) + err
>>> hermefit(x, y, 2)
array([ 1.01690445,  1.99951418,  2.99948696]) # may vary
```

`numpy.polynomial.hermite_e.`**`hermetrim`**(*c*, *tol=0*)

Remove "small" "trailing" coefficients from a polynomial.

"Small" means "small in absolute value" and is controlled by the parameter *tol*; "trailing" means highest order coefficient(s), e.g., in `[0, 1, 1, 0, 0]` (which represents `0 + x + x**2 + 0*x**3 + 0*x**4`) both the 3-rd and 4-th order coefficients would be "trimmed."

**Parameters**

**c**

[array_like] 1-d array of coefficients, ordered from lowest order to highest.

**tol**

[number, optional] Trailing (i.e., highest order) elements with absolute value less than or equal to *tol* (default value is zero) are removed.

**Returns**

**trimmed**

[ndarray] 1-d array with trailing zeros removed. If the resulting series would be empty, a series containing a single zero is returned.

**Raises**

**ValueError**

If $tol < 0$

**See also:**

`trimseq`

## Examples

```
>>> from numpy.polynomial import polyutils as pu
>>> pu.trimcoef((0,0,3,0,5,0,0))
array([0.,  0.,  3.,  0.,  5.])
>>> pu.trimcoef((0,0,1e-3,0,1e-5,0,0),1e-3) # item == tol is trimmed
array([0.])
>>> i = complex(0,1) # works for complex
>>> pu.trimcoef((3e-4,1e-3*(1-i),5e-4,2e-5*(1+i)), 1e-3)
array([0.0003+0.j   , 0.001 -0.001j])
```

`numpy.polynomial.hermite_e.`**`hermeline`**(*off*, *scl*)

Hermite series whose graph is a straight line.

**Parameters**

**off, scl**

[scalars] The specified line is given by `off + scl*x`.

**Returns**

**y**

[ndarray] This module's representation of the Hermite series for `off + scl*x`.

**See also:**

`polyline`, `chebline`

### Examples

```
>>> from numpy.polynomial.hermite_e import hermeline
>>> from numpy.polynomial.hermite_e import hermeline, hermeval
>>> hermeval(0,hermeline(3, 2))
3.0
>>> hermeval(1,hermeline(3, 2))
5.0
```

`numpy.polynomial.hermite_e.`**`herme2poly`**`(`*c*`)`

Convert a Hermite series to a polynomial.

Convert an array representing the coefficients of a Hermite series, ordered from lowest degree to highest, to an array of the coefficients of the equivalent polynomial (relative to the "standard" basis) ordered from lowest to highest degree.

**Parameters**

**c**

[array_like] 1-D array containing the Hermite series coefficients, ordered from lowest order term to highest.

**Returns**

**pol**

[ndarray] 1-D array containing the coefficients of the equivalent polynomial (relative to the "standard" basis) ordered from lowest order term to highest.

**See also:**

*poly2herme*

### Notes

The easy way to do conversions between polynomial basis sets is to use the convert method of a class instance.

### Examples

```
>>> from numpy.polynomial.hermite_e import herme2poly
>>> herme2poly([  2.,  10.,   2.,   3.])
array([0.,  1.,  2.,  3.])
```

`numpy.polynomial.hermite_e.`**`poly2herme`**`(`*pol*`)`

Convert a polynomial to a Hermite series.

Convert an array representing the coefficients of a polynomial (relative to the "standard" basis) ordered from lowest degree to highest, to an array of the coefficients of the equivalent Hermite series, ordered from lowest to highest degree.

**Parameters**

**pol**

[array_like] 1-D array containing the polynomial coefficients

**Returns**

**c**

[ndarray] 1-D array containing the coefficients of the equivalent Hermite series.

**See also:**

*herme2poly*

### Notes

The easy way to do conversions between polynomial basis sets is to use the convert method of a class instance.

### Examples

```
>>> from numpy.polynomial.hermite_e import poly2herme
>>> poly2herme(np.arange(4))
array([  2.,   10.,    2.,    3.])
```

### See also

`numpy.polynomial`

New in version 1.6.0.

## Laguerre Series (`numpy.polynomial.laguerre`)

This module provides a number of objects (mostly functions) useful for dealing with Laguerre series, including a *Laguerre* class that encapsulates the usual arithmetic operations. (General information on how this module represents and works with such polynomials is in the docstring for its "parent" sub-package, `numpy.polynomial`).

### Classes

| | |
|---|---|
| *Laguerre*(coef[, domain, window]) | A Laguerre series class. |

**class** `numpy.polynomial.laguerre.`**`Laguerre`**(*coef*, *domain=None*, *window=None*)

A Laguerre series class.

The Laguerre class provides the standard Python numerical methods '+', '-', '*', '//', '%', 'divmod', '**', and '()' as well as the attributes and methods listed in the `ABCPolyBase` documentation.

**Parameters**

**coef**

[array_like] Laguerre coefficients in order of increasing degree, i.e, `(1, 2, 3)` gives `1*L_0(x) + 2*L_1(X) + 3*L_2(x)`.

**domain**

[(2,) array_like, optional] Domain to use. The interval `[domain[0], domain[1]]` is mapped to the interval `[window[0], window[1]]` by shifting and scaling. The default value is [0, 1].

**window**

[(2,) array_like, optional] Window, see `domain` for its use. The default value is [0, 1].

New in version 1.6.0.

## Methods

| | |
|---|---|
| *`__call__`*(self, arg) | Call self as a function. |
| *`basis`*(deg[, domain, window]) | Series basis polynomial of degree *deg*. |
| *`cast`*(series[, domain, window]) | Convert series to series of this class. |
| *`convert`*(self[, domain, kind, window]) | Convert series to a different kind and/or domain and/or window. |
| *`copy`*(self) | Return a copy. |
| *`cutdeg`*(self, deg) | Truncate series to the given degree. |
| *`degree`*(self) | The degree of the series. |
| *`deriv`*(self[, m]) | Differentiate. |
| *`fit`*(x, y, deg[, domain, rcond, full, w, window]) | Least squares fit to data. |
| *`fromroots`*(roots[, domain, window]) | Return series instance that has the specified roots. |
| *`has_samecoef`*(self, other) | Check if coefficients match. |
| *`has_samedomain`*(self, other) | Check if domains match. |
| *`has_sametype`*(self, other) | Check if types match. |
| *`has_samewindow`*(self, other) | Check if windows match. |
| *`identity`*([domain, window]) | Identity function. |
| *`integ`*(self[, m, k, lbnd]) | Integrate. |
| *`linspace`*(self[, n, domain]) | Return x, y values at equally spaced points in domain. |
| *`mapparms`*(self) | Return the mapping parameters. |
| *`roots`*(self) | Return the roots of the series polynomial. |
| *`trim`*(self[, tol]) | Remove trailing coefficients |
| *`truncate`*(self, size) | Truncate series to length *size*. |

method

`Laguerre.`**`__call__`**(*self*, *arg*)
Call self as a function.

method

**`classmethod`** `Laguerre.`**`basis`**(*deg*, *domain=None*, *window=None*)
Series basis polynomial of degree *deg*.

Returns the series representing the basis polynomial of degree *deg*.

New in version 1.7.0.

**Parameters**

**deg**

[int] Degree of the basis polynomial for the series. Must be >= 0.

**domain**

[{None, array_like}, optional] If given, the array must be of the form `[beg, end]`, where `beg` and `end` are the endpoints of the domain. If None is given then the class domain is used. The default is None.

**window**

[{None, array_like}, optional] If given, the resulting array must be if the form `[beg, end]`, where `beg` and `end` are the endpoints of the window. If None is given then the class window is used. The default is None.

**Returns**

**new_series**

[series] A series with the coefficient of the *deg* term set to one and all others zero.

method

**`classmethod`** `Laguerre.`**`cast`**(*series*, *domain=None*, *window=None*)

Convert series to series of this class.

The *series* is expected to be an instance of some polynomial series of one of the types supported by by the numpy.polynomial module, but could be some other class that supports the convert method.

New in version 1.7.0.

**Parameters**

**series**

[series] The series instance to be converted.

**domain**

[{None, array_like}, optional] If given, the array must be of the form `[beg, end]`, where `beg` and `end` are the endpoints of the domain. If None is given then the class domain is used. The default is None.

**window**

[{None, array_like}, optional] If given, the resulting array must be if the form `[beg, end]`, where `beg` and `end` are the endpoints of the window. If None is given then the class window is used. The default is None.

**Returns**

**new_series**

[series] A series of the same kind as the calling class and equal to *series* when evaluated.

**See also:**

***`convert`***

similar instance method

method

`Laguerre.`**`convert`**(*self*, *domain=None*, *kind=None*, *window=None*)

Convert series to a different kind and/or domain and/or window.

**Parameters**

**domain**

[array_like, optional] The domain of the converted series. If the value is None, the default domain of *kind* is used.

**kind**

[class, optional] The polynomial series type class to which the current instance should be converted. If kind is None, then the class of the current instance is used.

**window**

[array_like, optional] The window of the converted series. If the value is None, the default window of *kind* is used.

**Returns**

**new_series**

[series] The returned class can be of different type than the current instance and/or have a different domain and/or different window.

### Notes

Conversion between domains and class types can result in numerically ill defined series.

method

`Laguerre.`**`copy`**(*self*)

Return a copy.

**Returns**

**new_series**

[series] Copy of self.

method

`Laguerre.`**`cutdeg`**(*self*, *deg*)

Truncate series to the given degree.

Reduce the degree of the series to *deg* by discarding the high order terms. If *deg* is greater than the current degree a copy of the current series is returned. This can be useful in least squares where the coefficients of the high degree terms may be very small.

New in version 1.5.0.

**Parameters**

**deg**

[non-negative int] The series is reduced to degree *deg* by discarding the high order terms. The value of *deg* must be a non-negative integer.

**Returns**

**new_series**

[series] New instance of series with reduced degree.

method

`Laguerre.`**`degree`**`(`*self*`)`

The degree of the series.

New in version 1.5.0.

**Returns**

**degree**

[int] Degree of the series, one less than the number of coefficients.

method

`Laguerre.`**`deriv`**`(`*self*, *m=1*`)`

Differentiate.

Return a series instance of that is the derivative of the current series.

**Parameters**

**m**

[non-negative int] Find the derivative of order *m*.

**Returns**

**new_series**

[series] A new series representing the derivative. The domain is the same as the domain of the differentiated series.

method

**`classmethod`** `Laguerre.`**`fit`**`(`*x*, *y*, *deg*, *domain=None*, *rcond=None*, *full=False*, *w=None*, *window=None*`)`

Least squares fit to data.

Return a series instance that is the least squares fit to the data *y* sampled at *x*. The domain of the returned instance can be specified and this will often result in a superior fit with less chance of ill conditioning.

**Parameters**

**x**

[array_like, shape (M,)] x-coordinates of the M sample points `(x[i], y[i])`.

**y**

[array_like, shape (M,) or (M, K)] y-coordinates of the sample points. Several data sets of sample points sharing the same x-coordinates can be fitted at once by passing in a 2D-array that contains one dataset per column.

**deg**

[int or 1-D array_like] Degree(s) of the fitting polynomials. If *deg* is a single integer all terms up to and including the *deg*'th term are included in the fit. For NumPy versions >= 1.11.0 a list of integers specifying the degrees of the terms to include may be used instead.

**domain**

[{None, [beg, end], []}, optional] Domain to use for the returned series. If `None`, then a minimal domain that covers the points *x* is chosen. If `[]` the class domain is used. The default value was the class domain in NumPy 1.4 and `None` in later versions. The `[]` option was added in numpy 1.5.0.

**rcond**

[float, optional] Relative condition number of the fit. Singular values smaller than this relative to the largest singular value will be ignored. The default value is len(x)*eps, where eps is the relative precision of the float type, about 2e-16 in most cases.

**full**

[bool, optional] Switch determining nature of return value. When it is False (the default) just the coefficients are returned, when True diagnostic information from the singular value decomposition is also returned.

**w**

[array_like, shape (M,), optional] Weights. If not None the contribution of each point `(x[i],y[i])` to the fit is weighted by *w[i]*. Ideally the weights are chosen so that the errors of the products `w[i]*y[i]` all have the same variance. The default value is None.

New in version 1.5.0.

**window**

[{[beg, end]}, optional] Window to use for the returned series. The default value is the default class domain

New in version 1.6.0.

**Returns**

**new_series**

[series] A series that represents the least squares fit to the data and has the domain and window specified in the call. If the coefficients for the unscaled and unshifted basis polynomials are of interest, do `new_series.convert().coef`.

**[resid, rank, sv, rcond]**

[list] These values are only returned if *full* = True

resid – sum of squared residuals of the least squares fit rank – the numerical rank of the scaled Vandermonde matrix sv – singular values of the scaled Vandermonde matrix rcond – value of *rcond*.

For more details, see *linalg.lstsq*.

method

**classmethod** Laguerre.**fromroots**(*roots*, *domain=[]*, *window=None*)

Return series instance that has the specified roots.

Returns a series representing the product `(x - r[0])*(x - r[1])*...*(x - r[n-1])`, where `r` is a list of roots.

**Parameters**

**roots**

[array_like] List of roots.

**domain**

[{[], None, array_like}, optional] Domain for the resulting series. If None the domain is the interval from the smallest root to the largest. If [] the domain is the class domain. The default is [].

**window**

[{None, array_like}, optional] Window for the returned series. If None the class window is used. The default is None.

**Returns**

**new_series**

[series] Series with the specified roots.

method

`Laguerre.`**`has_samecoef`**`(`*self*, *other*`)`

Check if coefficients match.

New in version 1.6.0.

**Parameters**

**other**

[class instance] The other class must have the `coef` attribute.

**Returns**

**bool**

[boolean] True if the coefficients are the same, False otherwise.

method

`Laguerre.`**`has_samedomain`**`(`*self*, *other*`)`

Check if domains match.

New in version 1.6.0.

**Parameters**

**other**

[class instance] The other class must have the `domain` attribute.

**Returns**

**bool**

[boolean] True if the domains are the same, False otherwise.

method

`Laguerre.`**`has_sametype`**`(`*self*, *other*`)`

Check if types match.

New in version 1.7.0.

**Parameters**

**other**

[object] Class instance.

**Returns**

**bool**

[boolean] True if other is same class as self

method

`Laguerre.`**`has_samewindow`**(*self*, *other*)

Check if windows match.

New in version 1.6.0.

**Parameters**

**other**

[class instance] The other class must have the `window` attribute.

**Returns**

**bool**

[boolean] True if the windows are the same, False otherwise.

method

**`classmethod`** `Laguerre.`**`identity`**(*domain=None*, *window=None*)

Identity function.

If `p` is the returned series, then `p(x) == x` for all values of x.

**Parameters**

**domain**

[{None, array_like}, optional] If given, the array must be of the form `[beg, end]`, where `beg` and `end` are the endpoints of the domain. If None is given then the class domain is used. The default is None.

**window**

[{None, array_like}, optional] If given, the resulting array must be if the form `[beg, end]`, where `beg` and `end` are the endpoints of the window. If None is given then the class window is used. The default is None.

**Returns**

**new_series**

[series] Series of representing the identity.

method

`Laguerre.`**`integ`**(*self*, *m=1*, *k=[]*, *lbnd=None*)

Integrate.

Return a series instance that is the definite integral of the current series.

**Parameters**

**m**

[non-negative int] The number of integrations to perform.

**k**

[array_like] Integration constants. The first constant is applied to the first integration, the second to the second, and so on. The list of values must less than or equal to *m* in length and any missing values are set to zero.

**lbnd**

[Scalar] The lower bound of the definite integral.

**Returns**

**new_series**

[series] A new series representing the integral. The domain is the same as the domain of the integrated series.

method

`Laguerre.`**`linspace`**(*self*, *n=100*, *domain=None*)

Return x, y values at equally spaced points in domain.

Returns the x, y values at *n* linearly spaced points across the domain. Here y is the value of the polynomial at the points x. By default the domain is the same as that of the series instance. This method is intended mostly as a plotting aid.

New in version 1.5.0.

**Parameters**

**n**

[int, optional] Number of point pairs to return. The default value is 100.

**domain**

[{None, array_like}, optional] If not None, the specified domain is used instead of that of the calling instance. It should be of the form `[beg,end]`. The default is None which case the class domain is used.

**Returns**

**x, y**

[ndarray] x is equal to linspace(self.domain[0], self.domain[1], n) and y is the series evaluated at element of x.

method

`Laguerre.`**`mapparms`**(*self*)

Return the mapping parameters.

The returned values define a linear map `off + scl*x` that is applied to the input arguments before the series is evaluated. The map depends on the `domain` and `window`; if the current `domain` is equal to the `window` the resulting map is the identity. If the coefficients of the series instance are to be used by themselves outside this class, then the linear function must be substituted for the `x` in the standard representation of the base polynomials.

**Returns**

**off, scl**

[float or complex] The mapping function is defined by `off + scl*x`.

### Notes

If the current domain is the interval `[l1, r1]` and the window is `[l2, r2]`, then the linear mapping function `L` is defined by the equations:

```
L(l1) = l2
L(r1) = r2
```

method

`Laguerre.`**`roots`**(*self*)

Return the roots of the series polynomial.

Compute the roots for the series. Note that the accuracy of the roots decrease the further outside the domain they lie.

**Returns**

**roots**

[ndarray] Array containing the roots of the series.

method

`Laguerre.`**`trim`**(*self*, *tol=0*)

Remove trailing coefficients

Remove trailing coefficients until a coefficient is reached whose absolute value greater than *tol* or the beginning of the series is reached. If all the coefficients would be removed the series is set to `[0]`. A new series instance is returned with the new coefficients. The current instance remains unchanged.

**Parameters**

**tol**

[non-negative number.] All trailing coefficients less than *tol* will be removed.

**Returns**

**new_series**

[series] Contains the new set of coefficients.

method

`Laguerre.`**`truncate`**`(`*self*, *size*`)`

Truncate series to length *size*.

Reduce the series to length *size* by discarding the high degree terms. The value of *size* must be a positive integer. This can be useful in least squares where the coefficients of the high degree terms may be very small.

**Parameters**

**size**

[positive int] The series is reduced to length *size* by discarding the high degree terms. The value of *size* must be a positive integer.

**Returns**

**new_series**

[series] New instance of series with truncated coefficients.

### Constants

| |
|---|
| *lagdomain* |
| *lagzero* |
| *lagone* |
| *lagx* |

`numpy.polynomial.laguerre.`**`lagdomain = array([0, 1])`**

`numpy.polynomial.laguerre.`**`lagzero = array([0])`**

`numpy.polynomial.laguerre.`**`lagone = array([1])`**

`numpy.polynomial.laguerre.`**`lagx = array([ 1, -1])`**

### Arithmetic

| | |
|---|---|
| *lagadd*(c1, c2) | Add one Laguerre series to another. |
| *lagsub*(c1, c2) | Subtract one Laguerre series from another. |
| *lagmulx*(c) | Multiply a Laguerre series by x. |
| *lagmul*(c1, c2) | Multiply one Laguerre series by another. |
| *lagdiv*(c1, c2) | Divide one Laguerre series by another. |
| *lagpow*(c, pow[, maxpower]) | Raise a Laguerre series to a power. |
| *lagval*(x, c[, tensor]) | Evaluate a Laguerre series at points x. |
| *lagval2d*(x, y, c) | Evaluate a 2-D Laguerre series at points (x, y). |
| *lagval3d*(x, y, z, c) | Evaluate a 3-D Laguerre series at points (x, y, z). |
| *laggrid2d*(x, y, c) | Evaluate a 2-D Laguerre series on the Cartesian product of x and y. |
| *laggrid3d*(x, y, z, c) | Evaluate a 3-D Laguerre series on the Cartesian product of x, y, and z. |

`numpy.polynomial.laguerre.`**`lagadd`**`(`*c1*, *c2*`)`

Add one Laguerre series to another.

Returns the sum of two Laguerre series *c1* + *c2*. The arguments are sequences of coefficients ordered from lowest order term to highest, i.e., [1,2,3] represents the series `P_0 + 2*P_1 + 3*P_2`.

**Parameters**

**c1, c2**

[array_like] 1-D arrays of Laguerre series coefficients ordered from low to high.

**Returns**

**out**

[ndarray] Array representing the Laguerre series of their sum.

**See also:**

*lagsub*, *lagmulx*, *lagmul*, *lagdiv*, *lagpow*

## Notes

Unlike multiplication, division, etc., the sum of two Laguerre series is a Laguerre series (without having to "reproject" the result onto the basis set) so addition, just like that of "standard" polynomials, is simply "component-wise."

## Examples

```
>>> from numpy.polynomial.laguerre import lagadd
>>> lagadd([1, 2, 3], [1, 2, 3, 4])
array([2.,  4.,  6.,  4.])
```

`numpy.polynomial.laguerre.`**`lagsub`**`(`*c1*, *c2*`)`

Subtract one Laguerre series from another.

Returns the difference of two Laguerre series *c1* - *c2*. The sequences of coefficients are from lowest order term to highest, i.e., [1,2,3] represents the series `P_0 + 2*P_1 + 3*P_2`.

**Parameters**

**c1, c2**

[array_like] 1-D arrays of Laguerre series coefficients ordered from low to high.

**Returns**

**out**

[ndarray] Of Laguerre series coefficients representing their difference.

**See also:**

*lagadd*, *lagmulx*, *lagmul*, *lagdiv*, *lagpow*

## Notes

Unlike multiplication, division, etc., the difference of two Laguerre series is a Laguerre series (without having to "reproject" the result onto the basis set) so subtraction, just like that of "standard" polynomials, is simply "component-wise."

## Examples

```
>>> from numpy.polynomial.laguerre import lagsub
>>> lagsub([1, 2, 3, 4], [1, 2, 3])
array([0.,  0.,  0.,  4.])
```

`numpy.polynomial.laguerre.`**`lagmulx`**`(`*c*`)`

Multiply a Laguerre series by x.

Multiply the Laguerre series *c* by x, where x is the independent variable.

**Parameters**

**c**

[array_like] 1-D array of Laguerre series coefficients ordered from low to high.

**Returns**

**out**

[ndarray] Array representing the result of the multiplication.

**See also:**

*lagadd*, *lagsub*, *lagmul*, *lagdiv*, *lagpow*

## Notes

The multiplication uses the recursion relationship for Laguerre polynomials in the form

xP_i(x) = (-(i + 1)*P_{i + 1}(x) + (2i + 1)P_{i}(x) - iP_{i - 1}(x))

## Examples

```
>>> from numpy.polynomial.laguerre import lagmulx
>>> lagmulx([1, 2, 3])
array([-1.,  -1.,  11.,  -9.])
```

`numpy.polynomial.laguerre.`**`lagmul`**`(`*c1*, *c2*`)`

Multiply one Laguerre series by another.

Returns the product of two Laguerre series *c1* * *c2*. The arguments are sequences of coefficients, from lowest order "term" to highest, e.g., [1,2,3] represents the series `P_0 + 2*P_1 + 3*P_2`.

**Parameters**

**c1, c2**

[array_like] 1-D arrays of Laguerre series coefficients ordered from low to high.

**Returns**

**out**

[ndarray] Of Laguerre series coefficients representing their product.

**See also:**

*lagadd*, *lagsub*, *lagmulx*, *lagdiv*, *lagpow*

## Notes

In general, the (polynomial) product of two C-series results in terms that are not in the Laguerre polynomial basis set. Thus, to express the product as a Laguerre series, it is necessary to “reproject” the product onto said basis set, which may produce “unintuitive” (but correct) results; see Examples section below.

## Examples

```
>>> from numpy.polynomial.laguerre import lagmul
>>> lagmul([1, 2, 3], [0, 1, 2])
array([  8., -13.,  38., -51.,  36.])
```

`numpy.polynomial.laguerre.`**`lagdiv`**`(`*c1*, *c2*`)`

Divide one Laguerre series by another.

Returns the quotient-with-remainder of two Laguerre series *c1* / *c2*. The arguments are sequences of coefficients from lowest order “term” to highest, e.g., [1,2,3] represents the series `P_0 + 2*P_1 + 3*P_2`.

**Parameters**

**c1, c2**

[array_like] 1-D arrays of Laguerre series coefficients ordered from low to high.

**Returns**

**[quo, rem]**

[ndarrays] Of Laguerre series coefficients representing the quotient and remainder.

**See also:**

*lagadd*, *lagsub*, *lagmulx*, *lagmul*, *lagpow*

## Notes

In general, the (polynomial) division of one Laguerre series by another results in quotient and remainder terms that are not in the Laguerre polynomial basis set. Thus, to express these results as a Laguerre series, it is necessary to “reproject” the results onto the Laguerre basis set, which may produce “unintuitive” (but correct) results; see Examples section below.

### Examples

```
>>> from numpy.polynomial.laguerre import lagdiv
>>> lagdiv([  8., -13.,  38., -51.,  36.], [0, 1, 2])
(array([1., 2., 3.]), array([0.]))
>>> lagdiv([  9., -12.,  38., -51.,  36.], [0, 1, 2])
(array([1., 2., 3.]), array([1., 1.]))
```

`numpy.polynomial.laguerre.`**`lagpow`**(*c*, *pow*, *maxpower=16*)

Raise a Laguerre series to a power.

Returns the Laguerre series *c* raised to the power *pow*. The argument *c* is a sequence of coefficients ordered from low to high. i.e., [1,2,3] is the series `P_0 + 2*P_1 + 3*P_2.`

**Parameters**

**c**

[array_like] 1-D array of Laguerre series coefficients ordered from low to high.

**pow**

[integer] Power to which the series will be raised

**maxpower**

[integer, optional] Maximum power allowed. This is mainly to limit growth of the series to unmanageable size. Default is 16

**Returns**

**coef**

[ndarray] Laguerre series of power.

**See also:**

*lagadd*, *lagsub*, *lagmulx*, *lagmul*, *lagdiv*

### Examples

```
>>> from numpy.polynomial.laguerre import lagpow
>>> lagpow([1, 2, 3], 2)
array([ 14., -16.,  56., -72.,  54.])
```

`numpy.polynomial.laguerre.`**`lagval`**(*x*, *c*, *tensor=True*)

Evaluate a Laguerre series at points x.

If *c* is of length *n + 1*, this function returns the value:

$$p(x) = c_0 * L_0(x) + c_1 * L_1(x) + ... + c_n * L_n(x)$$

The parameter *x* is converted to an array only if it is a tuple or a list, otherwise it is treated as a scalar. In either case, either *x* or its elements must support multiplication and addition both with themselves and with the elements of *c*.

If *c* is a 1-D array, then *p(x)* will have the same shape as *x*. If *c* is multidimensional, then the shape of the result depends on the value of *tensor*. If *tensor* is true the shape will be c.shape[1:] + x.shape. If *tensor* is false the shape will be c.shape[1:]. Note that scalars have shape (,).

Trailing zeros in the coefficients will be used in the evaluation, so they should be avoided if efficiency is a concern.

**Parameters**

**x**

[array_like, compatible object] If *x* is a list or tuple, it is converted to an ndarray, otherwise it is left unchanged and treated as a scalar. In either case, *x* or its elements must support addition and multiplication with with themselves and with the elements of *c*.

**c**

[array_like] Array of coefficients ordered so that the coefficients for terms of degree n are contained in c[n]. If *c* is multidimensional the remaining indices enumerate multiple polynomials. In the two dimensional case the coefficients may be thought of as stored in the columns of *c*.

**tensor**

[boolean, optional] If True, the shape of the coefficient array is extended with ones on the right, one for each dimension of *x*. Scalars have dimension 0 for this action. The result is that every column of coefficients in *c* is evaluated for every element of *x*. If False, *x* is broadcast over the columns of *c* for the evaluation. This keyword is useful when *c* is multidimensional. The default value is True.

New in version 1.7.0.

**Returns**

**values**

[ndarray, algebra_like] The shape of the return value is described above.

**See also:**

`lagval2d`, `laggrid2d`, `lagval3d`, `laggrid3d`

## Notes

The evaluation uses Clenshaw recursion, aka synthetic division.

## Examples

```
>>> from numpy.polynomial.laguerre import lagval
>>> coef = [1,2,3]
>>> lagval(1, coef)
-0.5
>>> lagval([[1,2],[3,4]], coef)
array([[-0.5, -4. ],
       [-4.5, -2. ]])
```

`numpy.polynomial.laguerre.`**`lagval2d`**`(x, y, c)`

Evaluate a 2-D Laguerre series at points (x, y).

This function returns the values:

$$p(x,y) = \sum_{i,j} c_{i,j} * L_i(x) * L_j(y)$$

The parameters *x* and *y* are converted to arrays only if they are tuples or a lists, otherwise they are treated as a scalars and they must have the same shape after conversion. In either case, either *x* and *y* or their elements must support multiplication and addition both with themselves and with the elements of *c*.

If *c* is a 1-D array a one is implicitly appended to its shape to make it 2-D. The shape of the result will be c.shape[2:] + x.shape.

**Parameters**

**x, y**

[array_like, compatible objects] The two dimensional series is evaluated at the points *(x, y)*, where *x* and *y* must have the same shape. If *x* or *y* is a list or tuple, it is first converted to an ndarray, otherwise it is left unchanged and if it isn't an ndarray it is treated as a scalar.

**c**

[array_like] Array of coefficients ordered so that the coefficient of the term of multi-degree i,j is contained in `c[i,j]`. If *c* has dimension greater than two the remaining indices enumerate multiple sets of coefficients.

**Returns**

**values**

[ndarray, compatible object] The values of the two dimensional polynomial at points formed with pairs of corresponding values from *x* and *y*.

**See also:**

`lagval`, `laggrid2d`, `lagval3d`, `laggrid3d`

## Notes

New in version 1.7.0.

`numpy.polynomial.laguerre.`**`lagval3d`**`(x, y, z, c)`

Evaluate a 3-D Laguerre series at points (x, y, z).

This function returns the values:

$$p(x,y,z) = \sum_{i,j,k} c_{i,j,k} * L_i(x) * L_j(y) * L_k(z)$$

The parameters *x*, *y*, and *z* are converted to arrays only if they are tuples or a lists, otherwise they are treated as a scalars and they must have the same shape after conversion. In either case, either *x*, *y*, and *z* or their elements must support multiplication and addition both with themselves and with the elements of *c*.

If *c* has fewer than 3 dimensions, ones are implicitly appended to its shape to make it 3-D. The shape of the result will be c.shape[3:] + x.shape.

**Parameters**

**x, y, z**

[array_like, compatible object] The three dimensional series is evaluated at the points *(x, y, z)*, where *x*, *y*, and *z* must have the same shape. If any of *x*, *y*, or *z* is a list or tuple, it is first converted to an ndarray, otherwise it is left unchanged and if it isn't an ndarray it is treated as a scalar.

**c**

[array_like] Array of coefficients ordered so that the coefficient of the term of multi-degree i,j,k is contained in `c[i,j,k]`. If *c* has dimension greater than 3 the remaining indices enumerate multiple sets of coefficients.

**Returns**

**values**

[ndarray, compatible object] The values of the multidimension polynomial on points formed with triples of corresponding values from *x*, *y*, and *z*.

**See also:**

*lagval*, *lagval2d*, *laggrid2d*, *laggrid3d*

## Notes

New in version 1.7.0.

`numpy.polynomial.laguerre.`**`laggrid2d`**`(`*x*, *y*, *c*`)`

Evaluate a 2-D Laguerre series on the Cartesian product of x and y.

This function returns the values:

$$p(a,b) = \sum_{i,j} c_{i,j} * L_i(a) * L_j(b)$$

where the points *(a, b)* consist of all pairs formed by taking *a* from *x* and *b* from *y*. The resulting points form a grid with *x* in the first dimension and *y* in the second.

The parameters *x* and *y* are converted to arrays only if they are tuples or a lists, otherwise they are treated as a scalars. In either case, either *x* and *y* or their elements must support multiplication and addition both with themselves and with the elements of *c*.

If *c* has fewer than two dimensions, ones are implicitly appended to its shape to make it 2-D. The shape of the result will be c.shape[2:] + x.shape + y.shape.

**Parameters**

**x, y**

[array_like, compatible objects] The two dimensional series is evaluated at the points in the Cartesian product of *x* and *y*. If *x* or *y* is a list or tuple, it is first converted to an ndarray, otherwise it is left unchanged and, if it isn't an ndarray, it is treated as a scalar.

**c**

[array_like] Array of coefficients ordered so that the coefficient of the term of multi-degree i,j is contained in *c[i,j]*. If *c* has dimension greater than two the remaining indices enumerate multiple sets of coefficients.

**Returns**

**values**

[ndarray, compatible object] The values of the two dimensional Chebyshev series at points in the Cartesian product of *x* and *y*.

**See also:**

*lagval*, *lagval2d*, *lagval3d*, *laggrid3d*

## Notes

New in version 1.7.0.

`numpy.polynomial.laguerre.`**`laggrid3d`**`(x, y, z, c)`

Evaluate a 3-D Laguerre series on the Cartesian product of x, y, and z.

This function returns the values:

$$p(a,b,c) = \sum_{i,j,k} c_{i,j,k} * L_i(a) * L_j(b) * L_k(c)$$

where the points *(a, b, c)* consist of all triples formed by taking *a* from *x*, *b* from *y*, and *c* from *z*. The resulting points form a grid with *x* in the first dimension, *y* in the second, and *z* in the third.

The parameters *x*, *y*, and *z* are converted to arrays only if they are tuples or a lists, otherwise they are treated as a scalars. In either case, either *x*, *y*, and *z* or their elements must support multiplication and addition both with themselves and with the elements of *c*.

If *c* has fewer than three dimensions, ones are implicitly appended to its shape to make it 3-D. The shape of the result will be c.shape[3:] + x.shape + y.shape + z.shape.

**Parameters**

**x, y, z**

[array_like, compatible objects] The three dimensional series is evaluated at the points in the Cartesian product of *x*, *y*, and *z*. If *x*,'y', or *z* is a list or tuple, it is first converted to an ndarray, otherwise it is left unchanged and, if it isn't an ndarray, it is treated as a scalar.

**c**

[array_like] Array of coefficients ordered so that the coefficients for terms of degree i,j are contained in `c[i,j]`. If *c* has dimension greater than two the remaining indices enumerate multiple sets of coefficients.

**Returns**

**values**

[ndarray, compatible object] The values of the two dimensional polynomial at points in the Cartesian product of *x* and *y*.

**See also:**

*lagval*, *lagval2d*, *laggrid2d*, *lagval3d*

### Notes

New in version 1.7.0.

## Calculus

| | |
|---|---|
| `lagder`(c[, m, scl, axis]) | Differentiate a Laguerre series. |
| `lagint`(c[, m, k, lbnd, scl, axis]) | Integrate a Laguerre series. |

`numpy.polynomial.laguerre.`**`lagder`**(*c*, *m=1*, *scl=1*, *axis=0*)

Differentiate a Laguerre series.

Returns the Laguerre series coefficients *c* differentiated *m* times along *axis*. At each iteration the result is multiplied by *scl* (the scaling factor is for use in a linear change of variable). The argument *c* is an array of coefficients from low to high degree along each axis, e.g., [1,2,3] represents the series `1*L_0 + 2*L_1 + 3*L_2` while [[1,2],[1,2]] represents `1*L_0(x)*L_0(y) + 1*L_1(x)*L_0(y) + 2*L_0(x)*L_1(y) + 2*L_1(x)*L_1(y)` if axis=0 is `x` and axis=1 is `y`.

**Parameters**

**c**

[array_like] Array of Laguerre series coefficients. If *c* is multidimensional the different axis correspond to different variables with the degree in each axis given by the corresponding index.

**m**

[int, optional] Number of derivatives taken, must be non-negative. (Default: 1)

**scl**

[scalar, optional] Each differentiation is multiplied by *scl*. The end result is multiplication by `scl**m`. This is for use in a linear change of variable. (Default: 1)

**axis**

[int, optional] Axis over which the derivative is taken. (Default: 0).

New in version 1.7.0.

**Returns**

**der**

[ndarray] Laguerre series of the derivative.

**See also:**

*`lagint`*

## Notes

In general, the result of differentiating a Laguerre series does not resemble the same operation on a power series. Thus the result of this function may be "unintuitive," albeit correct; see Examples section below.

## Examples

```
>>> from numpy.polynomial.laguerre import lagder
>>> lagder([ 1.,  1.,  1., -3.])
array([1.,  2.,  3.])
>>> lagder([ 1.,  0.,  0., -4.,  3.], m=2)
array([1.,  2.,  3.])
```

`numpy.polynomial.laguerre.`**`lagint`**(*c*, *m=1*, *k=[]*, *lbnd=0*, *scl=1*, *axis=0*)

Integrate a Laguerre series.

Returns the Laguerre series coefficients *c* integrated *m* times from *lbnd* along *axis*. At each iteration the resulting series is **multiplied** by *scl* and an integration constant, *k*, is added. The scaling factor is for use in a linear change of variable. ("Buyer beware": note that, depending on what one is doing, one may want *scl* to be the reciprocal of what one might expect; for more information, see the Notes section below.) The argument *c* is an array of coefficients from low to high degree along each axis, e.g., [1,2,3] represents the series `L_0 + 2*L_1 + 3*L_2` while [[1,2],[1,2]] represents `1*L_0(x)*L_0(y) + 1*L_1(x)*L_0(y) + 2*L_0(x)*L_1(y) + 2*L_1(x)*L_1(y)` if axis=0 is `x` and axis=1 is `y`.

**Parameters**

**c**

[array_like] Array of Laguerre series coefficients. If *c* is multidimensional the different axis correspond to different variables with the degree in each axis given by the corresponding index.

**m**

[int, optional] Order of integration, must be positive. (Default: 1)

**k**

[{[], list, scalar}, optional] Integration constant(s). The value of the first integral at `lbnd` is the first value in the list, the value of the second integral at `lbnd` is the second value, etc. If `k == []` (the default), all constants are set to zero. If `m == 1`, a single scalar can be given instead of a list.

**lbnd**

[scalar, optional] The lower bound of the integral. (Default: 0)

**scl**

[scalar, optional] Following each integration the result is *multiplied* by *scl* before the integration constant is added. (Default: 1)

**axis**

[int, optional] Axis over which the integral is taken. (Default: 0).

New in version 1.7.0.

**Returns**

**S**

[ndarray] Laguerre series coefficients of the integral.

**Raises**

**ValueError**

If `m < 0`, `len(k) > m`, `np.ndim(lbnd) != 0`, or `np.ndim(scl) != 0`.

**See also:**

`lagder`

## Notes

Note that the result of each integration is *multiplied* by *scl*. Why is this important to note? Say one is making a linear change of variable $u = ax + b$ in an integral relative to $x$. Then $dx = du/a$, so one will need to set *scl* equal to $1/a$ - perhaps not what one would have first thought.

Also note that, in general, the result of integrating a C-series needs to be "reprojected" onto the C-series basis set. Thus, typically, the result of this function is "unintuitive," albeit correct; see Examples section below.

## Examples

```
>>> from numpy.polynomial.laguerre import lagint
>>> lagint([1,2,3])
array([ 1.,  1.,  1., -3.])
>>> lagint([1,2,3], m=2)
array([ 1.,  0.,  0., -4.,  3.])
>>> lagint([1,2,3], k=1)
array([ 2.,  1.,  1., -3.])
>>> lagint([1,2,3], lbnd=-1)
array([11.5,  1. ,  1. , -3. ])
>>> lagint([1,2], m=2, k=[1,2], lbnd=-1)
array([ 11.16666667,  -5.        ,  -3.        ,   2.        ]) # may vary
```

## Misc Functions

| | |
|---|---|
| `lagfromroots`(roots) | Generate a Laguerre series with given roots. |
| `lagroots`(c) | Compute the roots of a Laguerre series. |
| `lagvander`(x, deg) | Pseudo-Vandermonde matrix of given degree. |
| `lagvander2d`(x, y, deg) | Pseudo-Vandermonde matrix of given degrees. |
| `lagvander3d`(x, y, z, deg) | Pseudo-Vandermonde matrix of given degrees. |
| `laggauss`(deg) | Gauss-Laguerre quadrature. |
| `lagweight`(x) | Weight function of the Laguerre polynomials. |
| `lagcompanion`(c) | Return the companion matrix of c. |
| `lagfit`(x, y, deg[, rcond, full, w]) | Least squares fit of Laguerre series to data. |
| `lagtrim`(c[, tol]) | Remove "small" "trailing" coefficients from a polynomial. |
| `lagline`(off, scl) | Laguerre series whose graph is a straight line. |
| `lag2poly`(c) | Convert a Laguerre series to a polynomial. |
| `poly2lag`(pol) | Convert a polynomial to a Laguerre series. |

`numpy.polynomial.laguerre.`**`lagfromroots`**`(`*roots*`)`

Generate a Laguerre series with given roots.

The function returns the coefficients of the polynomial

$$p(x) = (x - r_0) * (x - r_1) * ... * (x - r_n),$$

in Laguerre form, where the *r_n* are the roots specified in *roots*. If a zero has multiplicity n, then it must appear in *roots* n times. For instance, if 2 is a root of multiplicity three and 3 is a root of multiplicity 2, then *roots* looks something like [2, 2, 2, 3, 3]. The roots can appear in any order.

If the returned coefficients are *c*, then

$$p(x) = c_0 + c_1 * L_1(x) + ... + c_n * L_n(x)$$

The coefficient of the last term is not generally 1 for monic polynomials in Laguerre form.

**Parameters**

**roots**

[array_like] Sequence containing the roots.

**Returns**

**out**

[ndarray] 1-D array of coefficients. If all roots are real then *out* is a real array, if some of the roots are complex, then *out* is complex even if all the coefficients in the result are real (see Examples below).

**See also:**

`polyfromroots`, `legfromroots`, `chebfromroots`, `hermfromroots`, `hermefromroots`

### Examples

```
>>> from numpy.polynomial.laguerre import lagfromroots, lagval
>>> coef = lagfromroots((-1, 0, 1))
>>> lagval((-1, 0, 1), coef)
array([0.,  0.,  0.])
>>> coef = lagfromroots((-1j, 1j))
>>> lagval((-1j, 1j), coef)
array([0.+0.j, 0.+0.j])
```

`numpy.polynomial.laguerre.`**`lagroots`**`(c)`

Compute the roots of a Laguerre series.

Return the roots (a.k.a. “zeros”) of the polynomial

$$p(x) = \sum_i c[i] * L_i(x).$$

**Parameters**

**c**

[1-D array_like] 1-D array of coefficients.

**Returns**

**out**

[ndarray] Array of the roots of the series. If all the roots are real, then *out* is also real, otherwise it is complex.

**See also:**

`polyroots`, `legroots`, `chebroots`, `hermroots`, `hermeroots`

## Notes

The root estimates are obtained as the eigenvalues of the companion matrix, Roots far from the origin of the complex plane may have large errors due to the numerical instability of the series for such values. Roots with multiplicity greater than 1 will also show larger errors as the value of the series near such points is relatively insensitive to errors in the roots. Isolated roots near the origin can be improved by a few iterations of Newton's method.

The Laguerre series basis polynomials aren't powers of $x$ so the results of this function may seem unintuitive.

## Examples

```
>>> from numpy.polynomial.laguerre import lagroots, lagfromroots
>>> coef = lagfromroots([0, 1, 2])
>>> coef
array([  2.,  -8.,  12.,  -6.])
>>> lagroots(coef)
array([-4.4408921e-16,  1.0000000e+00,  2.0000000e+00])
```

`numpy.polynomial.laguerre.`**`lagvander`**`(`*x*, *deg*`)`

Pseudo-Vandermonde matrix of given degree.

Returns the pseudo-Vandermonde matrix of degree *deg* and sample points *x*. The pseudo-Vandermonde matrix is defined by

$$V[..., i] = L_i(x)$$

where *0* <= *i* <= *deg*. The leading indices of *V* index the elements of *x* and the last index is the degree of the Laguerre polynomial.

If *c* is a 1-D array of coefficients of length *n* + *1* and *V* is the array `V = lagvander(x, n)`, then `np.dot(V, c)` and `lagval(x, c)` are the same up to roundoff. This equivalence is useful both for least squares fitting and for the evaluation of a large number of Laguerre series of the same degree and sample points.

**Parameters**

**x**

[array_like] Array of points. The dtype is converted to float64 or complex128 depending on whether any of the elements are complex. If *x* is scalar it is converted to a 1-D array.

**deg**

[int] Degree of the resulting matrix.

**Returns**

**vander**

[ndarray] The pseudo-Vandermonde matrix. The shape of the returned matrix is `x.shape + (deg + 1,)`, where The last index is the degree of the corresponding Laguerre polynomial. The dtype will be the same as the converted *x*.

## Examples

```
>>> from numpy.polynomial.laguerre import lagvander
>>> x = np.array([0, 1, 2])
>>> lagvander(x, 3)
array([[ 1.        ,  1.        ,  1.        ,  1.        ],
       [ 1.        ,  0.        , -0.5       , -0.66666667],
       [ 1.        , -1.        , -1.        , -0.33333333]])
```

`numpy.polynomial.laguerre.`**`lagvander2d`**(*x*, *y*, *deg*)

Pseudo-Vandermonde matrix of given degrees.

Returns the pseudo-Vandermonde matrix of degrees *deg* and sample points *(x, y)*. The pseudo-Vandermonde matrix is defined by

$$V[..., (deg[1] + 1) * i + j] = L_i(x) * L_j(y),$$

where *0 <= i <= deg[0]* and *0 <= j <= deg[1]*. The leading indices of *V* index the points *(x, y)* and the last index encodes the degrees of the Laguerre polynomials.

If `V = lagvander2d(x, y, [xdeg, ydeg])`, then the columns of *V* correspond to the elements of a 2-D coefficient array *c* of shape (xdeg + 1, ydeg + 1) in the order

$$c_{00}, c_{01}, c_{02}..., c_{10}, c_{11}, c_{12}...$$

and `np.dot(V, c.flat)` and `lagval2d(x, y, c)` will be the same up to roundoff. This equivalence is useful both for least squares fitting and for the evaluation of a large number of 2-D Laguerre series of the same degrees and sample points.

**Parameters**

**x, y**

[array_like] Arrays of point coordinates, all of the same shape. The dtypes will be converted to either float64 or complex128 depending on whether any of the elements are complex. Scalars are converted to 1-D arrays.

**deg**

[list of ints] List of maximum degrees of the form [x_deg, y_deg].

**Returns**

**vander2d**

[ndarray] The shape of the returned matrix is `x.shape + (order,)`, where $order = (deg[0] + 1) * (deg[1] + 1)$. The dtype will be the same as the converted *x* and *y*.

**See also:**

*`lagvander`*, *`lagvander3d`*, *`lagval2d`*, *`lagval3d`*

### Notes

New in version 1.7.0.

`numpy.polynomial.laguerre.`**`lagvander3d`**`(`*x*, *y*, *z*, *deg*`)`

Pseudo-Vandermonde matrix of given degrees.

Returns the pseudo-Vandermonde matrix of degrees *deg* and sample points *(x, y, z)*. If *l, m, n* are the given degrees in *x, y, z*, then The pseudo-Vandermonde matrix is defined by

$$V[..., (m+1)(n+1)i + (n+1)j + k] = L_i(x) * L_j(y) * L_k(z),$$

where *0 <= i <= l*, *0 <= j <= m*, and *0 <= j <= n*. The leading indices of $V$ index the points *(x, y, z)* and the last index encodes the degrees of the Laguerre polynomials.

If `V = lagvander3d(x, y, z, [xdeg, ydeg, zdeg])`, then the columns of $V$ correspond to the elements of a 3-D coefficient array *c* of shape (xdeg + 1, ydeg + 1, zdeg + 1) in the order

$$c_{000}, c_{001}, c_{002}, ..., c_{010}, c_{011}, c_{012}, ...$$

and `np.dot(V, c.flat)` and `lagval3d(x, y, z, c)` will be the same up to roundoff. This equivalence is useful both for least squares fitting and for the evaluation of a large number of 3-D Laguerre series of the same degrees and sample points.

**Parameters**

**x, y, z**

[array_like] Arrays of point coordinates, all of the same shape. The dtypes will be converted to either float64 or complex128 depending on whether any of the elements are complex. Scalars are converted to 1-D arrays.

**deg**

[list of ints] List of maximum degrees of the form [x_deg, y_deg, z_deg].

**Returns**

**vander3d**

[ndarray] The shape of the returned matrix is `x.shape + (order,)`, where $order = (deg[0]+1) * (deg[1]+1) * (deg[2]+1)$. The dtype will be the same as the converted *x*, *y*, and *z*.

**See also:**

*`lagvander`*, *`lagvander3d`*, *`lagval2d`*, *`lagval3d`*

### Notes

New in version 1.7.0.

`numpy.polynomial.laguerre.`**`laggauss`**`(`*deg*`)`

Gauss-Laguerre quadrature.

Computes the sample points and weights for Gauss-Laguerre quadrature. These sample points and weights will correctly integrate polynomials of degree $2 * deg - 1$ or less over the interval $[0, \inf]$ with the weight function $f(x) = \exp(-x)$.

**Parameters**

**deg**

[int] Number of sample points and weights. It must be >= 1.

**Returns**

**x**

[ndarray] 1-D ndarray containing the sample points.

**y**

[ndarray] 1-D ndarray containing the weights.

## Notes

New in version 1.7.0.

The results have only been tested up to degree 100 higher degrees may be problematic. The weights are determined by using the fact that

$$w_k = c/(L'_n(x_k) * L_{n-1}(x_k))$$

where $c$ is a constant independent of $k$ and $x_k$ is the k'th root of $L_n$, and then scaling the results to get the right value when integrating 1.

`numpy.polynomial.laguerre.`**`lagweight`**`(x)`

Weight function of the Laguerre polynomials.

The weight function is $exp(-x)$ and the interval of integration is $[0, \text{inf}]$. The Laguerre polynomials are orthogonal, but not normalized, with respect to this weight function.

**Parameters**

**x**

[array_like] Values at which the weight function will be computed.

**Returns**

**w**

[ndarray] The weight function at *x*.

## Notes

New in version 1.7.0.

`numpy.polynomial.laguerre.`**`lagcompanion`**`(c)`

Return the companion matrix of c.

The usual companion matrix of the Laguerre polynomials is already symmetric when *c* is a basis Laguerre polynomial, so no scaling is applied.

**Parameters**

**c**

[array_like] 1-D array of Laguerre series coefficients ordered from low to high degree.

**Returns**

**mat**

[ndarray] Companion matrix of dimensions (deg, deg).

### Notes

New in version 1.7.0.

`numpy.polynomial.laguerre.`**`lagfit`**(*x*, *y*, *deg*, *rcond=None*, *full=False*, *w=None*)

Least squares fit of Laguerre series to data.

Return the coefficients of a Laguerre series of degree *deg* that is the least squares fit to the data values *y* given at points *x*. If *y* is 1-D the returned coefficients will also be 1-D. If *y* is 2-D multiple fits are done, one for each column of *y*, and the resulting coefficients are stored in the corresponding columns of a 2-D return. The fitted polynomial(s) are in the form

$$p(x) = c_0 + c_1 * L_1(x) + ... + c_n * L_n(x),$$

where *n* is *deg*.

**Parameters**

**x**

[array_like, shape (M,)] x-coordinates of the M sample points `(x[i], y[i])`.

**y**

[array_like, shape (M,) or (M, K)] y-coordinates of the sample points. Several data sets of sample points sharing the same x-coordinates can be fitted at once by passing in a 2D-array that contains one dataset per column.

**deg**

[int or 1-D array_like] Degree(s) of the fitting polynomials. If *deg* is a single integer all terms up to and including the *deg*'th term are included in the fit. For NumPy versions >= 1.11.0 a list of integers specifying the degrees of the terms to include may be used instead.

**rcond**

[float, optional] Relative condition number of the fit. Singular values smaller than this relative to the largest singular value will be ignored. The default value is len(x)*eps, where eps is the relative precision of the float type, about 2e-16 in most cases.

**full**

[bool, optional] Switch determining nature of return value. When it is False (the default) just the coefficients are returned, when True diagnostic information from the singular value decomposition is also returned.

**w**

[array_like, shape (*M*,), optional] Weights. If not None, the contribution of each point `(x[i],y[i])` to the fit is weighted by *w[i]*. Ideally the weights are chosen so that the errors of the products `w[i]*y[i]` all have the same variance. The default value is None.

**Returns**

**coef**

[ndarray, shape (M,) or (M, K)] Laguerre coefficients ordered from low to high. If *y* was 2-D, the coefficients for the data in column k of *y* are in column *k*.

**[residuals, rank, singular_values, rcond]**

[list] These values are only returned if *full* = True

resid – sum of squared residuals of the least squares fit rank – the numerical rank of the scaled Vandermonde matrix sv – singular values of the scaled Vandermonde matrix rcond – value of *rcond*.

For more details, see *linalg.lstsq*.

**Warns**

**RankWarning**

The rank of the coefficient matrix in the least-squares fit is deficient. The warning is only raised if *full* = False. The warnings can be turned off by

```
>>> import warnings
>>> warnings.simplefilter('ignore', np.RankWarning)
```

**See also:**

`chebfit`, `legfit`, `polyfit`, `hermfit`, `hermefit`

**`lagval`**

Evaluates a Laguerre series.

**`lagvander`**

pseudo Vandermonde matrix of Laguerre series.

**`lagweight`**

Laguerre weight function.

**`linalg.lstsq`**

Computes a least-squares fit from the matrix.

**`scipy.interpolate.UnivariateSpline`**

Computes spline fits.

## Notes

The solution is the coefficients of the Laguerre series *p* that minimizes the sum of the weighted squared errors

$$E = \sum_j w_j^2 * |y_j - p(x_j)|^2,$$

where the $w_j$ are the weights. This problem is solved by setting up as the (typically) overdetermined matrix equation

$$V(x) * c = w * y,$$

where *V* is the weighted pseudo Vandermonde matrix of *x*, *c* are the coefficients to be solved for, *w* are the weights, and *y* are the observed values. This equation is then solved using the singular value decomposition of *V*.

If some of the singular values of $V$ are so small that they are neglected, then a *RankWarning* will be issued. This means that the coefficient values may be poorly determined. Using a lower order fit will usually get rid of the warning. The *rcond* parameter can also be set to a value smaller than its default, but the resulting fit may be spurious and have large contributions from roundoff error.

Fits using Laguerre series are probably most useful when the data can be approximated by `sqrt(w(x)) * p(x)`, where *w(x)* is the Laguerre weight. In that case the weight `sqrt(w(x[i]))` should be used together with data values `y[i]/sqrt(w(x[i]))`. The weight function is available as `lagweight`.

### References

[1]

### Examples

```
>>> from numpy.polynomial.laguerre import lagfit, lagval
>>> x = np.linspace(0, 10)
>>> err = np.random.randn(len(x))/10
>>> y = lagval(x, [1, 2, 3]) + err
>>> lagfit(x, y, 2)
array([ 0.96971004,  2.00193749,  3.00288744]) # may vary
```

`numpy.polynomial.laguerre.`**`lagtrim`**`(`*c*, *tol=0*`)`

Remove "small" "trailing" coefficients from a polynomial.

"Small" means "small in absolute value" and is controlled by the parameter *tol*; "trailing" means highest order coefficient(s), e.g., in `[0, 1, 1, 0, 0]` (which represents `0 + x + x**2 + 0*x**3 + 0*x**4`) both the 3-rd and 4-th order coefficients would be "trimmed."

**Parameters**

**c**

[array_like] 1-d array of coefficients, ordered from lowest order to highest.

**tol**

[number, optional] Trailing (i.e., highest order) elements with absolute value less than or equal to *tol* (default value is zero) are removed.

**Returns**

**trimmed**

[ndarray] 1-d array with trailing zeros removed. If the resulting series would be empty, a series containing a single zero is returned.

**Raises**

**ValueError**

If $tol < 0$

**See also:**

`trimseq`

### Examples

```
>>> from numpy.polynomial import polyutils as pu
>>> pu.trimcoef((0,0,3,0,5,0,0))
array([0.,  0.,  3.,  0.,  5.])
>>> pu.trimcoef((0,0,1e-3,0,1e-5,0,0),1e-3) # item == tol is trimmed
array([0.])
>>> i = complex(0,1) # works for complex
>>> pu.trimcoef((3e-4,1e-3*(1-i),5e-4,2e-5*(1+i)), 1e-3)
array([0.0003+0.j   , 0.001 -0.001j])
```

`numpy.polynomial.laguerre.`**`lagline`**`(`*off*, *scl*`)`

Laguerre series whose graph is a straight line.

**Parameters**

**off, scl**

[scalars] The specified line is given by `off + scl*x`.

**Returns**

**y**

[ndarray] This module's representation of the Laguerre series for `off + scl*x`.

**See also:**

`polyline`, `chebline`

### Examples

```
>>> from numpy.polynomial.laguerre import lagline, lagval
>>> lagval(0,lagline(3, 2))
3.0
>>> lagval(1,lagline(3, 2))
5.0
```

`numpy.polynomial.laguerre.`**`lag2poly`**`(`*c*`)`

Convert a Laguerre series to a polynomial.

Convert an array representing the coefficients of a Laguerre series, ordered from lowest degree to highest, to an array of the coefficients of the equivalent polynomial (relative to the "standard" basis) ordered from lowest to highest degree.

**Parameters**

**c**

[array_like] 1-D array containing the Laguerre series coefficients, ordered from lowest order term to highest.

**Returns**

**pol**

[ndarray] 1-D array containing the coefficients of the equivalent polynomial (relative to the "standard" basis) ordered from lowest order term to highest.

**See also:**

*poly2lag*

## Notes

The easy way to do conversions between polynomial basis sets is to use the convert method of a class instance.

## Examples

```
>>> from numpy.polynomial.laguerre import lag2poly
>>> lag2poly([ 23., -63.,  58., -18.])
array([0., 1., 2., 3.])
```

`numpy.polynomial.laguerre.`**`poly2lag`**`(`*pol*`)`

Convert a polynomial to a Laguerre series.

Convert an array representing the coefficients of a polynomial (relative to the “standard” basis) ordered from lowest degree to highest, to an array of the coefficients of the equivalent Laguerre series, ordered from lowest to highest degree.

**Parameters**

**pol**

[array_like] 1-D array containing the polynomial coefficients

**Returns**

**c**

[ndarray] 1-D array containing the coefficients of the equivalent Laguerre series.

**See also:**

*lag2poly*

## Notes

The easy way to do conversions between polynomial basis sets is to use the convert method of a class instance.

## Examples

```
>>> from numpy.polynomial.laguerre import poly2lag
>>> poly2lag(np.arange(4))
array([ 23., -63.,  58., -18.])
```

### See also

`numpy.polynomial`

New in version 1.6.0.

## Legendre Series (`numpy.polynomial.legendre`)

This module provides a number of objects (mostly functions) useful for dealing with Legendre series, including a `Legendre` class that encapsulates the usual arithmetic operations. (General information on how this module represents and works with such polynomials is in the docstring for its "parent" sub-package, `numpy.polynomial`).

### Classes

| | |
|---|---|
| `Legendre`(coef[, domain, window]) | A Legendre series class. |

**class** `numpy.polynomial.legendre.`**`Legendre`**(*coef*, *domain=None*, *window=None*)

A Legendre series class.

The Legendre class provides the standard Python numerical methods '+', '-', '*', '//', '%', 'divmod', '**', and '()' as well as the attributes and methods listed in the `ABCPolyBase` documentation.

**Parameters**

**coef**

[array_like] Legendre coefficients in order of increasing degree, i.e., `(1, 2, 3)` gives `1*P_0(x) + 2*P_1(x) + 3*P_2(x)`.

**domain**

[(2,) array_like, optional] Domain to use. The interval `[domain[0], domain[1]]` is mapped to the interval `[window[0], window[1]]` by shifting and scaling. The default value is [-1, 1].

**window**

[(2,) array_like, optional] Window, see `domain` for its use. The default value is [-1, 1].

New in version 1.6.0.

### Methods

| | |
|---|---|
| `__call__`(self, arg) | Call self as a function. |
| `basis`(deg[, domain, window]) | Series basis polynomial of degree *deg*. |
| `cast`(series[, domain, window]) | Convert series to series of this class. |
| `convert`(self[, domain, kind, window]) | Convert series to a different kind and/or domain and/or window. |
| `copy`(self) | Return a copy. |
| `cutdeg`(self, deg) | Truncate series to the given degree. |
| `degree`(self) | The degree of the series. |
| `deriv`(self[, m]) | Differentiate. |
| `fit`(x, y, deg[, domain, rcond, full, w, window]) | Least squares fit to data. |
| `fromroots`(roots[, domain, window]) | Return series instance that has the specified roots. |
| `has_samecoef`(self, other) | Check if coefficients match. |

Continued on next page

Table 128 – continued from previous page

| | |
|---|---|
| `has_samedomain`(self, other) | Check if domains match. |
| `has_sametype`(self, other) | Check if types match. |
| `has_samewindow`(self, other) | Check if windows match. |
| `identity`([domain, window]) | Identity function. |
| `integ`(self[, m, k, lbnd]) | Integrate. |
| `linspace`(self[, n, domain]) | Return x, y values at equally spaced points in domain. |
| `mapparms`(self) | Return the mapping parameters. |
| `roots`(self) | Return the roots of the series polynomial. |
| `trim`(self[, tol]) | Remove trailing coefficients |
| `truncate`(self, size) | Truncate series to length *size*. |

method

`Legendre.`**`__call__`**(*self*, *arg*)

Call self as a function.

method

**`classmethod`** `Legendre.`**`basis`**(*deg*, *domain=None*, *window=None*)

Series basis polynomial of degree *deg*.

Returns the series representing the basis polynomial of degree *deg*.

New in version 1.7.0.

**Parameters**

**deg**

[int] Degree of the basis polynomial for the series. Must be >= 0.

**domain**

[{None, array_like}, optional] If given, the array must be of the form `[beg, end]`, where `beg` and `end` are the endpoints of the domain. If None is given then the class domain is used. The default is None.

**window**

[{None, array_like}, optional] If given, the resulting array must be if the form `[beg, end]`, where `beg` and `end` are the endpoints of the window. If None is given then the class window is used. The default is None.

**Returns**

**new_series**

[series] A series with the coefficient of the *deg* term set to one and all others zero.

method

**`classmethod`** `Legendre.`**`cast`**(*series*, *domain=None*, *window=None*)

Convert series to series of this class.

The *series* is expected to be an instance of some polynomial series of one of the types supported by by the numpy.polynomial module, but could be some other class that supports the convert method.

New in version 1.7.0.

**Parameters**

**series**

[series] The series instance to be converted.

**domain**

[{None, array_like}, optional] If given, the array must be of the form `[beg, end]`, where `beg` and `end` are the endpoints of the domain. If None is given then the class domain is used. The default is None.

**window**

[{None, array_like}, optional] If given, the resulting array must be if the form `[beg, end]`, where `beg` and `end` are the endpoints of the window. If None is given then the class window is used. The default is None.

**Returns**

**new_series**

[series] A series of the same kind as the calling class and equal to *series* when evaluated.

**See also:**

***`convert`***

similar instance method

method

`Legendre.`**`convert`**(*self*, *domain=None*, *kind=None*, *window=None*)

Convert series to a different kind and/or domain and/or window.

**Parameters**

**domain**

[array_like, optional] The domain of the converted series. If the value is None, the default domain of *kind* is used.

**kind**

[class, optional] The polynomial series type class to which the current instance should be converted. If kind is None, then the class of the current instance is used.

**window**

[array_like, optional] The window of the converted series. If the value is None, the default window of *kind* is used.

**Returns**

**new_series**

[series] The returned class can be of different type than the current instance and/or have a different domain and/or different window.

### Notes

Conversion between domains and class types can result in numerically ill defined series.

method

`Legendre.`**`copy`**`(`*self*`)`

Return a copy.

**Returns**

**new_series**

[series] Copy of self.

method

`Legendre.`**`cutdeg`**`(`*self*, *deg*`)`

Truncate series to the given degree.

Reduce the degree of the series to *deg* by discarding the high order terms. If *deg* is greater than the current degree a copy of the current series is returned. This can be useful in least squares where the coefficients of the high degree terms may be very small.

New in version 1.5.0.

**Parameters**

**deg**

[non-negative int] The series is reduced to degree *deg* by discarding the high order terms. The value of *deg* must be a non-negative integer.

**Returns**

**new_series**

[series] New instance of series with reduced degree.

method

`Legendre.`**`degree`**`(`*self*`)`

The degree of the series.

New in version 1.5.0.

**Returns**

**degree**

[int] Degree of the series, one less than the number of coefficients.

method

`Legendre.`**`deriv`**`(`*self*, *m=1*`)`

Differentiate.

Return a series instance of that is the derivative of the current series.

**Parameters**

**m**

[non-negative int] Find the derivative of order *m*.

**Returns**

**new_series**

[series] A new series representing the derivative. The domain is the same as the domain of the differentiated series.

method

**classmethod** `Legendre.`**`fit`**(*x*, *y*, *deg*, *domain=None*, *rcond=None*, *full=False*, *w=None*, *window=None*)

Least squares fit to data.

Return a series instance that is the least squares fit to the data *y* sampled at *x*. The domain of the returned instance can be specified and this will often result in a superior fit with less chance of ill conditioning.

**Parameters**

**x**

[array_like, shape (M,)] x-coordinates of the M sample points `(x[i], y[i])`.

**y**

[array_like, shape (M,) or (M, K)] y-coordinates of the sample points. Several data sets of sample points sharing the same x-coordinates can be fitted at once by passing in a 2D-array that contains one dataset per column.

**deg**

[int or 1-D array_like] Degree(s) of the fitting polynomials. If *deg* is a single integer all terms up to and including the *deg*'th term are included in the fit. For NumPy versions >= 1.11.0 a list of integers specifying the degrees of the terms to include may be used instead.

**domain**

[{None, [beg, end], []}, optional] Domain to use for the returned series. If `None`, then a minimal domain that covers the points *x* is chosen. If `[]` the class domain is used. The default value was the class domain in NumPy 1.4 and `None` in later versions. The `[]` option was added in numpy 1.5.0.

**rcond**

[float, optional] Relative condition number of the fit. Singular values smaller than this relative to the largest singular value will be ignored. The default value is len(x)*eps, where eps is the relative precision of the float type, about 2e-16 in most cases.

**full**

[bool, optional] Switch determining nature of return value. When it is False (the default) just the coefficients are returned, when True diagnostic information from the singular value decomposition is also returned.

**w**

[array_like, shape (M,), optional] Weights. If not None the contribution of each point `(x[i],y[i])` to the fit is weighted by *w[i]*. Ideally the weights are chosen so that the errors of the products `w[i]*y[i]` all have the same variance. The default value is None.

New in version 1.5.0.

**window**

[{[beg, end]}, optional] Window to use for the returned series. The default value is the default class domain

New in version 1.6.0.

**Returns**

**new_series**

[series] A series that represents the least squares fit to the data and has the domain and window specified in the call. If the coefficients for the unscaled and unshifted basis polynomials are of interest, do `new_series.convert().coef`.

**[resid, rank, sv, rcond]**

[list] These values are only returned if *full* = True

resid – sum of squared residuals of the least squares fit rank – the numerical rank of the scaled Vandermonde matrix sv – singular values of the scaled Vandermonde matrix rcond – value of *rcond*.

For more details, see *linalg.lstsq*.

method

**classmethod** `Legendre.`**`fromroots`**`(`*roots*, *domain=[]*, *window=None*`)`

Return series instance that has the specified roots.

Returns a series representing the product `(x - r[0])*(x - r[1])*...*(x - r[n-1])`, where `r` is a list of roots.

**Parameters**

**roots**

[array_like] List of roots.

**domain**

[{[], None, array_like}, optional] Domain for the resulting series. If None the domain is the interval from the smallest root to the largest. If [] the domain is the class domain. The default is [].

**window**

[{None, array_like}, optional] Window for the returned series. If None the class window is used. The default is None.

**Returns**

**new_series**

[series] Series with the specified roots.

method

`Legendre.`**`has_samecoef`**`(`*self*, *other*`)`

Check if coefficients match.

New in version 1.6.0.

**Parameters**

**other**

[class instance] The other class must have the `coef` attribute.

**Returns**

**bool**

[boolean] True if the coefficients are the same, False otherwise.

method

`Legendre.`**`has_samedomain`**(*self*, *other*)

Check if domains match.

New in version 1.6.0.

**Parameters**

**other**

[class instance] The other class must have the `domain` attribute.

**Returns**

**bool**

[boolean] True if the domains are the same, False otherwise.

method

`Legendre.`**`has_sametype`**(*self*, *other*)

Check if types match.

New in version 1.7.0.

**Parameters**

**other**

[object] Class instance.

**Returns**

**bool**

[boolean] True if other is same class as self

method

`Legendre.`**`has_samewindow`**(*self*, *other*)

Check if windows match.

New in version 1.6.0.

**Parameters**

**other**

[class instance] The other class must have the `window` attribute.

**Returns**

**bool**

[boolean] True if the windows are the same, False otherwise.

method

**classmethod** `Legendre.`**`identity`**(*domain=None*, *window=None*)

Identity function.

If `p` is the returned series, then `p(x) == x` for all values of x.

**Parameters**

**domain**

[{None, array_like}, optional] If given, the array must be of the form `[beg, end]`, where `beg` and `end` are the endpoints of the domain. If None is given then the class domain is used. The default is None.

**window**

[{None, array_like}, optional] If given, the resulting array must be if the form `[beg, end]`, where `beg` and `end` are the endpoints of the window. If None is given then the class window is used. The default is None.

**Returns**

**new_series**

[series] Series of representing the identity.

method

`Legendre.`**`integ`**(*self*, *m=1*, *k=[]*, *lbnd=None*)

Integrate.

Return a series instance that is the definite integral of the current series.

**Parameters**

**m**

[non-negative int] The number of integrations to perform.

**k**

[array_like] Integration constants. The first constant is applied to the first integration, the second to the second, and so on. The list of values must less than or equal to *m* in length and any missing values are set to zero.

**lbnd**

[Scalar] The lower bound of the definite integral.

**Returns**

**new_series**

[series] A new series representing the integral. The domain is the same as the domain of the integrated series.

method

`Legendre.`**`linspace`**(*self*, *n=100*, *domain=None*)

Return x, y values at equally spaced points in domain.

Returns the x, y values at *n* linearly spaced points across the domain. Here y is the value of the polynomial at the points x. By default the domain is the same as that of the series instance. This method is intended mostly as a plotting aid.

New in version 1.5.0.

**Parameters**

**n**

[int, optional] Number of point pairs to return. The default value is 100.

**domain**

[{None, array_like}, optional] If not None, the specified domain is used instead of that of the calling instance. It should be of the form `[beg,end]`. The default is None which case the class domain is used.

**Returns**

**x, y**

[ndarray] x is equal to linspace(self.domain[0], self.domain[1], n) and y is the series evaluated at element of `x`.

method

`Legendre.`**`mapparms`**(*self*)

Return the mapping parameters.

The returned values define a linear map `off + scl*x` that is applied to the input arguments before the series is evaluated. The map depends on the `domain` and `window`; if the current `domain` is equal to the `window` the resulting map is the identity. If the coefficients of the series instance are to be used by themselves outside this class, then the linear function must be substituted for the `x` in the standard representation of the base polynomials.

**Returns**

**off, scl**

[float or complex] The mapping function is defined by `off + scl*x`.

### Notes

If the current domain is the interval `[l1, r1]` and the window is `[l2, r2]`, then the linear mapping function `L` is defined by the equations:

```
L(l1) = l2
L(r1) = r2
```

method

`Legendre.`**`roots`**(*self*)

Return the roots of the series polynomial.

Compute the roots for the series. Note that the accuracy of the roots decrease the further outside the domain they lie.

**Returns**

**roots**

[ndarray] Array containing the roots of the series.

method

`Legendre.`**`trim`**`(`*self*, *tol=0*`)`

Remove trailing coefficients

Remove trailing coefficients until a coefficient is reached whose absolute value greater than *tol* or the beginning of the series is reached. If all the coefficients would be removed the series is set to `[0]`. A new series instance is returned with the new coefficients. The current instance remains unchanged.

**Parameters**

**tol**

[non-negative number.] All trailing coefficients less than *tol* will be removed.

**Returns**

**new_series**

[series] Contains the new set of coefficients.

method

`Legendre.`**`truncate`**`(`*self*, *size*`)`

Truncate series to length *size*.

Reduce the series to length *size* by discarding the high degree terms. The value of *size* must be a positive integer. This can be useful in least squares where the coefficients of the high degree terms may be very small.

**Parameters**

**size**

[positive int] The series is reduced to length *size* by discarding the high degree terms. The value of *size* must be a positive integer.

**Returns**

**new_series**

[series] New instance of series with truncated coefficients.

### Constants

| |
|---|
| *legdomain* |
| *legzero* |
| *legone* |
| *legx* |

`numpy.polynomial.legendre.`**`legdomain = array([-1, 1])`**

`numpy.polynomial.legendre.`**`legzero = array([0])`**

`numpy.polynomial.legendre.`**`legone = array([1])`**

`numpy.polynomial.legendre.`**`legx = array([0, 1])`**

### Arithmetic

| | |
|---|---|
| *legadd*(c1, c2) | Add one Legendre series to another. |
| *legsub*(c1, c2) | Subtract one Legendre series from another. |
| *legmulx*(c) | Multiply a Legendre series by x. |
| *legmul*(c1, c2) | Multiply one Legendre series by another. |
| *legdiv*(c1, c2) | Divide one Legendre series by another. |
| *legpow*(c, pow[, maxpower]) | Raise a Legendre series to a power. |
| *legval*(x, c[, tensor]) | Evaluate a Legendre series at points x. |
| *legval2d*(x, y, c) | Evaluate a 2-D Legendre series at points (x, y). |
| *legval3d*(x, y, z, c) | Evaluate a 3-D Legendre series at points (x, y, z). |
| *leggrid2d*(x, y, c) | Evaluate a 2-D Legendre series on the Cartesian product of x and y. |
| *leggrid3d*(x, y, z, c) | Evaluate a 3-D Legendre series on the Cartesian product of x, y, and z. |

`numpy.polynomial.legendre.`**`legadd`**`(`*c1*, *c2*`)`

Add one Legendre series to another.

Returns the sum of two Legendre series $c1 + c2$. The arguments are sequences of coefficients ordered from lowest order term to highest, i.e., [1,2,3] represents the series `P_0 + 2*P_1 + 3*P_2`.

**Parameters**

**c1, c2**

[array_like] 1-D arrays of Legendre series coefficients ordered from low to high.

**Returns**

**out**

[ndarray] Array representing the Legendre series of their sum.

**See also:**

*legsub*, *legmulx*, *legmul*, *legdiv*, *legpow*

### Notes

Unlike multiplication, division, etc., the sum of two Legendre series is a Legendre series (without having to "reproject" the result onto the basis set) so addition, just like that of "standard" polynomials, is simply "component-wise."

### Examples

```
>>> from numpy.polynomial import legendre as L
>>> c1 = (1,2,3)
>>> c2 = (3,2,1)
>>> L.legadd(c1,c2)
array([4.,  4.,  4.])
```

`numpy.polynomial.legendre.`**`legsub`**`(`*c1*, *c2*`)`

Subtract one Legendre series from another.

Returns the difference of two Legendre series *c1* - *c2*. The sequences of coefficients are from lowest order term to highest, i.e., [1,2,3] represents the series `P_0 + 2*P_1 + 3*P_2`.

**Parameters**

**c1, c2**

[array_like] 1-D arrays of Legendre series coefficients ordered from low to high.

**Returns**

**out**

[ndarray] Of Legendre series coefficients representing their difference.

**See also:**

*`legadd`*, *`legmulx`*, *`legmul`*, *`legdiv`*, *`legpow`*

### Notes

Unlike multiplication, division, etc., the difference of two Legendre series is a Legendre series (without having to "reproject" the result onto the basis set) so subtraction, just like that of "standard" polynomials, is simply "component-wise."

### Examples

```
>>> from numpy.polynomial import legendre as L
>>> c1 = (1,2,3)
>>> c2 = (3,2,1)
>>> L.legsub(c1,c2)
array([-2.,  0.,  2.])
>>> L.legsub(c2,c1) # -C.legsub(c1,c2)
array([ 2.,  0., -2.])
```

`numpy.polynomial.legendre.`**`legmulx`**`(`*c*`)`

Multiply a Legendre series by x.

Multiply the Legendre series *c* by x, where x is the independent variable.

**Parameters**

**c**

[array_like] 1-D array of Legendre series coefficients ordered from low to high.

**Returns**

**out**

[ndarray] Array representing the result of the multiplication.

**See also:**

*legadd*, *legmul*, *legmul*, *legdiv*, *legpow*

## Notes

The multiplication uses the recursion relationship for Legendre polynomials in the form

$$xP_i(x) = ((i+1) * P_{i+1}(x) + i * P_{i-1}(x))/(2i+1)$$

## Examples

```
>>> from numpy.polynomial import legendre as L
>>> L.legmulx([1,2,3])
array([ 0.66666667, 2.2, 1.33333333, 1.8]) # may vary
```

`numpy.polynomial.legendre.`**`legmul`**`(`*c1*, *c2*`)`

Multiply one Legendre series by another.

Returns the product of two Legendre series *c1* * *c2*. The arguments are sequences of coefficients, from lowest order "term" to highest, e.g., [1,2,3] represents the series `P_0 + 2*P_1 + 3*P_2`.

**Parameters**

**c1, c2**

[array_like] 1-D arrays of Legendre series coefficients ordered from low to high.

**Returns**

**out**

[ndarray] Of Legendre series coefficients representing their product.

**See also:**

*legadd*, *legsub*, *legmulx*, *legdiv*, *legpow*

## Notes

In general, the (polynomial) product of two C-series results in terms that are not in the Legendre polynomial basis set. Thus, to express the product as a Legendre series, it is necessary to "reproject" the product onto said basis set, which may produce "unintuitive" (but correct) results; see Examples section below.

## Examples

```
>>> from numpy.polynomial import legendre as L
>>> c1 = (1,2,3)
>>> c2 = (3,2)
>>> L.legmul(c1,c2) # multiplication requires "reprojection"
array([  4.33333333,  10.4       ,  11.66666667,   3.6       ]) # may vary
```

`numpy.polynomial.legendre.`**`legdiv`**`(`*c1*, *c2*`)`

Divide one Legendre series by another.

Returns the quotient-with-remainder of two Legendre series *c1* / *c2*. The arguments are sequences of coefficients from lowest order "term" to highest, e.g., [1,2,3] represents the series `P_0 + 2*P_1 + 3*P_2`.

**Parameters**

**c1, c2**

[array_like] 1-D arrays of Legendre series coefficients ordered from low to high.

**Returns**

**quo, rem**

[ndarrays] Of Legendre series coefficients representing the quotient and remainder.

**See also:**

*`legadd`*, *`legsub`*, *`legmulx`*, *`legmul`*, *`legpow`*

## Notes

In general, the (polynomial) division of one Legendre series by another results in quotient and remainder terms that are not in the Legendre polynomial basis set. Thus, to express these results as a Legendre series, it is necessary to "reproject" the results onto the Legendre basis set, which may produce "unintuitive" (but correct) results; see Examples section below.

## Examples

```
>>> from numpy.polynomial import legendre as L
>>> c1 = (1,2,3)
>>> c2 = (3,2,1)
>>> L.legdiv(c1,c2) # quotient "intuitive," remainder not
(array([3.]), array([-8., -4.]))
>>> c2 = (0,1,2,3)
>>> L.legdiv(c2,c1) # neither "intuitive"
(array([-0.07407407,  1.66666667]), array([-1.03703704, -2.51851852])) # may vary
```

`numpy.polynomial.legendre.`**`legpow`**(*c*, *pow*, *maxpower=16*)

Raise a Legendre series to a power.

Returns the Legendre series *c* raised to the power *pow*. The argument *c* is a sequence of coefficients ordered from low to high. i.e., [1,2,3] is the series `P_0 + 2*P_1 + 3*P_2.`

**Parameters**

**c**

[array_like] 1-D array of Legendre series coefficients ordered from low to high.

**pow**

[integer] Power to which the series will be raised

**maxpower**

[integer, optional] Maximum power allowed. This is mainly to limit growth of the series to unmanageable size. Default is 16

**Returns**

**coef**

[ndarray] Legendre series of power.

**See also:**

`legadd`, `legsub`, `legmulx`, `legmul`, `legdiv`

`numpy.polynomial.legendre.`**`legval`**(*x*, *c*, *tensor=True*)

Evaluate a Legendre series at points x.

If *c* is of length *n + 1*, this function returns the value:

$$p(x) = c_0 * L_0(x) + c_1 * L_1(x) + ... + c_n * L_n(x)$$

The parameter *x* is converted to an array only if it is a tuple or a list, otherwise it is treated as a scalar. In either case, either *x* or its elements must support multiplication and addition both with themselves and with the elements of *c*.

If *c* is a 1-D array, then *p(x)* will have the same shape as *x*. If *c* is multidimensional, then the shape of the result depends on the value of *tensor*. If *tensor* is true the shape will be c.shape[1:] + x.shape. If *tensor* is false the shape will be c.shape[1:]. Note that scalars have shape (,).

Trailing zeros in the coefficients will be used in the evaluation, so they should be avoided if efficiency is a concern.

**Parameters**

**x**

[array_like, compatible object] If *x* is a list or tuple, it is converted to an ndarray, otherwise it is left unchanged and treated as a scalar. In either case, *x* or its elements must support addition and multiplication with with themselves and with the elements of *c*.

**c**

[array_like] Array of coefficients ordered so that the coefficients for terms of degree n are contained in c[n]. If *c* is multidimensional the remaining indices enumerate multiple polynomials. In the two dimensional case the coefficients may be thought of as stored in the columns of *c*.

**tensor**

[boolean, optional] If True, the shape of the coefficient array is extended with ones on the right, one for each dimension of *x*. Scalars have dimension 0 for this action. The result is that every column of coefficients in *c* is evaluated for every element of *x*. If False, *x* is broadcast over the columns of *c* for the evaluation. This keyword is useful when *c* is multidimensional. The default value is True.

New in version 1.7.0.

**Returns**

**values**

[ndarray, algebra_like] The shape of the return value is described above.

**See also:**

`legval2d`, `leggrid2d`, `legval3d`, `leggrid3d`

## Notes

The evaluation uses Clenshaw recursion, aka synthetic division.

`numpy.polynomial.legendre.`**`legval2d`**(*x*, *y*, *c*)

Evaluate a 2-D Legendre series at points (x, y).

This function returns the values:

$$p(x,y) = \sum_{i,j} c_{i,j} * L_i(x) * L_j(y)$$

The parameters *x* and *y* are converted to arrays only if they are tuples or a lists, otherwise they are treated as a scalars and they must have the same shape after conversion. In either case, either *x* and *y* or their elements must support multiplication and addition both with themselves and with the elements of *c*.

If *c* is a 1-D array a one is implicitly appended to its shape to make it 2-D. The shape of the result will be c.shape[2:] + x.shape.

**Parameters**

**x, y**

[array_like, compatible objects] The two dimensional series is evaluated at the points *(x, y)*, where *x* and *y* must have the same shape. If *x* or *y* is a list or tuple, it is first converted to an ndarray, otherwise it is left unchanged and if it isn't an ndarray it is treated as a scalar.

**c**

[array_like] Array of coefficients ordered so that the coefficient of the term of multi-degree i,j is contained in `c[i,j]`. If *c* has dimension greater than two the remaining indices enumerate multiple sets of coefficients.

**Returns**

**values**

[ndarray, compatible object] The values of the two dimensional Legendre series at points formed from pairs of corresponding values from *x* and *y*.

**See also:**

*legval*, *leggrid2d*, *legval3d*, *leggrid3d*

## Notes

New in version 1.7.0.

`numpy.polynomial.legendre.`**`legval3d`**`(`*x*, *y*, *z*, *c*`)`

Evaluate a 3-D Legendre series at points (x, y, z).

This function returns the values:

$$p(x,y,z) = \sum_{i,j,k} c_{i,j,k} * L_i(x) * L_j(y) * L_k(z)$$

The parameters *x*, *y*, and *z* are converted to arrays only if they are tuples or a lists, otherwise they are treated as a scalars and they must have the same shape after conversion. In either case, either *x*, *y*, and *z* or their elements must support multiplication and addition both with themselves and with the elements of *c*.

If *c* has fewer than 3 dimensions, ones are implicitly appended to its shape to make it 3-D. The shape of the result will be c.shape[3:] + x.shape.

**Parameters**

**x, y, z**

[array_like, compatible object] The three dimensional series is evaluated at the points *(x, y, z)*, where *x*, *y*, and *z* must have the same shape. If any of *x*, *y*, or *z* is a list or tuple, it is first converted to an ndarray, otherwise it is left unchanged and if it isn't an ndarray it is treated as a scalar.

**c**

[array_like] Array of coefficients ordered so that the coefficient of the term of multi-degree i,j,k is contained in `c[i,j,k]`. If *c* has dimension greater than 3 the remaining indices enumerate multiple sets of coefficients.

**Returns**

**values**

[ndarray, compatible object] The values of the multidimensional polynomial on points formed with triples of corresponding values from *x*, *y*, and *z*.

**See also:**

*legval*, *legval2d*, *leggrid2d*, *leggrid3d*

## Notes

New in version 1.7.0.

`numpy.polynomial.legendre.`**`leggrid2d`**`(x, y, c)`

Evaluate a 2-D Legendre series on the Cartesian product of x and y.

This function returns the values:

$$p(a,b) = \sum_{i,j} c_{i,j} * L_i(a) * L_j(b)$$

where the points *(a, b)* consist of all pairs formed by taking *a* from *x* and *b* from *y*. The resulting points form a grid with *x* in the first dimension and *y* in the second.

The parameters *x* and *y* are converted to arrays only if they are tuples or a lists, otherwise they are treated as a scalars. In either case, either *x* and *y* or their elements must support multiplication and addition both with themselves and with the elements of *c*.

If *c* has fewer than two dimensions, ones are implicitly appended to its shape to make it 2-D. The shape of the result will be c.shape[2:] + x.shape + y.shape.

**Parameters**

**x, y**

[array_like, compatible objects] The two dimensional series is evaluated at the points in the Cartesian product of *x* and *y*. If *x* or *y* is a list or tuple, it is first converted to an ndarray, otherwise it is left unchanged and, if it isn't an ndarray, it is treated as a scalar.

**c**

[array_like] Array of coefficients ordered so that the coefficient of the term of multi-degree i,j is contained in *c[i,j]*. If *c* has dimension greater than two the remaining indices enumerate multiple sets of coefficients.

**Returns**

**values**

[ndarray, compatible object] The values of the two dimensional Chebyshev series at points in the Cartesian product of *x* and *y*.

**See also:**

*legval*, *legval2d*, *legval3d*, *leggrid3d*

## Notes

New in version 1.7.0.

`numpy.polynomial.legendre.`**`leggrid3d`**`(x, y, z, c)`

Evaluate a 3-D Legendre series on the Cartesian product of x, y, and z.

This function returns the values:

$$p(a,b,c) = \sum_{i,j,k} c_{i,j,k} * L_i(a) * L_j(b) * L_k(c)$$

where the points *(a, b, c)* consist of all triples formed by taking *a* from *x*, *b* from *y*, and *c* from *z*. The resulting points form a grid with *x* in the first dimension, *y* in the second, and *z* in the third.

The parameters *x*, *y*, and *z* are converted to arrays only if they are tuples or a lists, otherwise they are treated as a scalars. In either case, either *x*, *y*, and *z* or their elements must support multiplication and addition both with themselves and with the elements of *c*.

If *c* has fewer than three dimensions, ones are implicitly appended to its shape to make it 3-D. The shape of the result will be c.shape[3:] + x.shape + y.shape + z.shape.

**Parameters**

**x, y, z**

[array_like, compatible objects] The three dimensional series is evaluated at the points in the Cartesian product of *x*, *y*, and *z*. If *x*,'y', or *z* is a list or tuple, it is first converted to an ndarray, otherwise it is left unchanged and, if it isn't an ndarray, it is treated as a scalar.

**c**

[array_like] Array of coefficients ordered so that the coefficients for terms of degree i,j are contained in `c[i,j]`. If *c* has dimension greater than two the remaining indices enumerate multiple sets of coefficients.

**Returns**

**values**

[ndarray, compatible object] The values of the two dimensional polynomial at points in the Cartesian product of *x* and *y*.

**See also:**

*legval*, *legval2d*, *leggrid2d*, *legval3d*

### Notes

New in version 1.7.0.

## Calculus

| | |
|---|---|
| *legder*(c[, m, scl, axis]) | Differentiate a Legendre series. |
| *legint*(c[, m, k, lbnd, scl, axis]) | Integrate a Legendre series. |

`numpy.polynomial.legendre.`**`legder`**(*c*, *m=1*, *scl=1*, *axis=0*)

Differentiate a Legendre series.

Returns the Legendre series coefficients *c* differentiated *m* times along *axis*. At each iteration the result is multiplied by *scl* (the scaling factor is for use in a linear change of variable). The argument *c* is an array of coefficients from low to high degree along each axis, e.g., [1,2,3] represents the series `1*L_0 + 2*L_1 + 3*L_2` while [[1,2],[1,2]] represents `1*L_0(x)*L_0(y) + 1*L_1(x)*L_0(y) + 2*L_0(x)*L_1(y) + 2*L_1(x)*L_1(y)` if axis=0 is `x` and axis=1 is `y`.

**Parameters**

**c**

[array_like] Array of Legendre series coefficients. If c is multidimensional the different axis correspond to different variables with the degree in each axis given by the corresponding index.

**m**

[int, optional] Number of derivatives taken, must be non-negative. (Default: 1)

**scl**

[scalar, optional] Each differentiation is multiplied by *scl*. The end result is multiplication by `scl**m`. This is for use in a linear change of variable. (Default: 1)

**axis**

[int, optional] Axis over which the derivative is taken. (Default: 0).

New in version 1.7.0.

**Returns**

**der**

[ndarray] Legendre series of the derivative.

**See also:**

*legint*

### Notes

In general, the result of differentiating a Legendre series does not resemble the same operation on a power series. Thus the result of this function may be "unintuitive," albeit correct; see Examples section below.

### Examples

```
>>> from numpy.polynomial import legendre as L
>>> c = (1,2,3,4)
>>> L.legder(c)
array([  6.,   9.,  20.])
>>> L.legder(c, 3)
array([60.])
>>> L.legder(c, scl=-1)
array([ -6.,  -9., -20.])
>>> L.legder(c, 2,-1)
array([  9.,  60.])
```

`numpy.polynomial.legendre.`**`legint`**(*c*, *m=1*, *k=[]*, *lbnd=0*, *scl=1*, *axis=0*)

Integrate a Legendre series.

Returns the Legendre series coefficients *c* integrated *m* times from *lbnd* along *axis*. At each iteration the resulting series is **multiplied** by *scl* and an integration constant, *k*, is added. The scaling factor is for use in a linear change of variable. ("Buyer beware": note that, depending on what one is doing, one may want *scl* to be the reciprocal of what one might expect; for more information, see the Notes section below.) The argument *c* is an array of coefficients from low to high degree along each axis, e.g., [1,2,3] represents the series `L_0 + 2*L_1 + 3*L_2` while [[1,2],[1,2]] represents `1*L_0(x)*L_0(y) + 1*L_1(x)*L_0(y) + 2*L_0(x)*L_1(y) + 2*L_1(x)*L_1(y)` if axis=0 is `x` and axis=1 is `y`.

**Parameters**

**c**

[array_like] Array of Legendre series coefficients. If c is multidimensional the different axis correspond to different variables with the degree in each axis given by the corresponding index.

**m**

[int, optional] Order of integration, must be positive. (Default: 1)

**k**

[{[], list, scalar}, optional] Integration constant(s). The value of the first integral at `lbnd` is the first value in the list, the value of the second integral at `lbnd` is the second value, etc. If `k == []` (the default), all constants are set to zero. If `m == 1`, a single scalar can be given instead of a list.

**lbnd**

[scalar, optional] The lower bound of the integral. (Default: 0)

**scl**

[scalar, optional] Following each integration the result is *multiplied* by *scl* before the integration constant is added. (Default: 1)

**axis**

[int, optional] Axis over which the integral is taken. (Default: 0).

New in version 1.7.0.

**Returns**

**S**

[ndarray] Legendre series coefficient array of the integral.

**Raises**

**ValueError**

If `m < 0`, `len(k) > m`, `np.ndim(lbnd) != 0`, or `np.ndim(scl) != 0`.

**See also:**

*`legder`*

### Notes

Note that the result of each integration is *multiplied* by *scl*. Why is this important to note? Say one is making a linear change of variable $u = ax + b$ in an integral relative to $x$. Then $dx = du/a$, so one will need to set *scl* equal to $1/a$ - perhaps not what one would have first thought.

Also note that, in general, the result of integrating a C-series needs to be "reprojected" onto the C-series basis set. Thus, typically, the result of this function is "unintuitive," albeit correct; see Examples section below.

### Examples

```
>>> from numpy.polynomial import legendre as L
>>> c = (1,2,3)
>>> L.legint(c)
array([ 0.33333333,  0.4       ,  0.66666667,  0.6       ]) # may vary
>>> L.legint(c, 3)
array([  1.66666667e-02,  -1.78571429e-02,   4.76190476e-02, # may vary
         -1.73472348e-18,   1.90476190e-02,   9.52380952e-03])
>>> L.legint(c, k=3)
 array([ 3.33333333,  0.4       ,  0.66666667,  0.6       ]) # may vary
>>> L.legint(c, lbnd=-2)
array([ 7.33333333,  0.4       ,  0.66666667,  0.6       ]) # may vary
>>> L.legint(c, scl=2)
array([ 0.66666667,  0.8       ,  1.33333333,  1.2       ]) # may vary
```

## Misc Functions

| | |
|---|---|
| *legfromroots*(roots) | Generate a Legendre series with given roots. |
| *legroots*(c) | Compute the roots of a Legendre series. |
| *legvander*(x, deg) | Pseudo-Vandermonde matrix of given degree. |
| *legvander2d*(x, y, deg) | Pseudo-Vandermonde matrix of given degrees. |
| *legvander3d*(x, y, z, deg) | Pseudo-Vandermonde matrix of given degrees. |
| *leggauss*(deg) | Gauss-Legendre quadrature. |
| *legweight*(x) | Weight function of the Legendre polynomials. |
| *legcompanion*(c) | Return the scaled companion matrix of c. |
| *legfit*(x, y, deg[, rcond, full, w]) | Least squares fit of Legendre series to data. |
| *legtrim*(c[, tol]) | Remove "small" "trailing" coefficients from a polynomial. |
| *legline*(off, scl) | Legendre series whose graph is a straight line. |
| *leg2poly*(c) | Convert a Legendre series to a polynomial. |
| *poly2leg*(pol) | Convert a polynomial to a Legendre series. |

`numpy.polynomial.legendre.`**`legfromroots`**(*roots*)

Generate a Legendre series with given roots.

The function returns the coefficients of the polynomial

$$p(x) = (x - r_0) * (x - r_1) * ... * (x - r_n),$$

in Legendre form, where the *r_n* are the roots specified in *roots*. If a zero has multiplicity n, then it must appear in *roots* n times. For instance, if 2 is a root of multiplicity three and 3 is a root of multiplicity 2, then *roots* looks something like [2, 2, 2, 3, 3]. The roots can appear in any order.

If the returned coefficients are *c*, then

$$p(x) = c_0 + c_1 * L_1(x) + ... + c_n * L_n(x)$$

The coefficient of the last term is not generally 1 for monic polynomials in Legendre form.

**Parameters**

**roots**

[array_like] Sequence containing the roots.

**Returns**

**out**

[ndarray] 1-D array of coefficients. If all roots are real then *out* is a real array, if some of the roots are complex, then *out* is complex even if all the coefficients in the result are real (see Examples below).

**See also:**

`polyfromroots`, `chebfromroots`, `lagfromroots`, `hermfromroots`, `hermefromroots`

## Examples

```
>>> import numpy.polynomial.legendre as L
>>> L.legfromroots((-1,0,1)) # x^3 - x relative to the standard basis
array([ 0. , -0.4,  0. ,  0.4])
>>> j = complex(0,1)
>>> L.legfromroots((-j,j)) # x^2 + 1 relative to the standard basis
array([ 1.33333333+0.j,  0.00000000+0.j,  0.66666667+0.j]) # may vary
```

`numpy.polynomial.legendre.`**`legroots`**`(c)`

Compute the roots of a Legendre series.

Return the roots (a.k.a. "zeros") of the polynomial

$$p(x) = \sum_i c[i] * L_i(x).$$

**Parameters**

**c**

[1-D array_like] 1-D array of coefficients.

**Returns**

**out**

[ndarray] Array of the roots of the series. If all the roots are real, then *out* is also real, otherwise it is complex.

**See also:**

`polyroots`, `chebroots`, `lagroots`, `hermroots`, `hermeroots`

## Notes

The root estimates are obtained as the eigenvalues of the companion matrix, Roots far from the origin of the complex plane may have large errors due to the numerical instability of the series for such values. Roots with multiplicity greater than 1 will also show larger errors as the value of the series near such points is relatively insensitive to errors in the roots. Isolated roots near the origin can be improved by a few iterations of Newton's method.

The Legendre series basis polynomials aren't powers of `x` so the results of this function may seem unintuitive.

### Examples

```
>>> import numpy.polynomial.legendre as leg
>>> leg.legroots((1, 2, 3, 4)) # 4L_3 + 3L_2 + 2L_1 + 1L_0, all real roots
array([-0.85099543, -0.11407192,  0.51506735]) # may vary
```

`numpy.polynomial.legendre.`**`legvander`**(*x*, *deg*)

Pseudo-Vandermonde matrix of given degree.

Returns the pseudo-Vandermonde matrix of degree *deg* and sample points *x*. The pseudo-Vandermonde matrix is defined by

$$V[..., i] = L_i(x)$$

where *0 <= i <= deg*. The leading indices of *V* index the elements of *x* and the last index is the degree of the Legendre polynomial.

If *c* is a 1-D array of coefficients of length *n + 1* and *V* is the array `V = legvander(x, n)`, then `np.dot(V, c)` and `legval(x, c)` are the same up to roundoff. This equivalence is useful both for least squares fitting and for the evaluation of a large number of Legendre series of the same degree and sample points.

**Parameters**

**x**

[array_like] Array of points. The dtype is converted to float64 or complex128 depending on whether any of the elements are complex. If *x* is scalar it is converted to a 1-D array.

**deg**

[int] Degree of the resulting matrix.

**Returns**

**vander**

[ndarray] The pseudo-Vandermonde matrix. The shape of the returned matrix is `x.shape + (deg + 1,)`, where The last index is the degree of the corresponding Legendre polynomial. The dtype will be the same as the converted *x*.

`numpy.polynomial.legendre.`**`legvander2d`**(*x*, *y*, *deg*)

Pseudo-Vandermonde matrix of given degrees.

Returns the pseudo-Vandermonde matrix of degrees *deg* and sample points *(x, y)*. The pseudo-Vandermonde matrix is defined by

$$V[..., (deg[1] + 1) * i + j] = L_i(x) * L_j(y),$$

where *0 <= i <= deg[0]* and *0 <= j <= deg[1]*. The leading indices of *V* index the points *(x, y)* and the last index encodes the degrees of the Legendre polynomials.

If `V = legvander2d(x, y, [xdeg, ydeg])`, then the columns of *V* correspond to the elements of a 2-D coefficient array *c* of shape (xdeg + 1, ydeg + 1) in the order

$$c_{00}, c_{01}, c_{02}..., c_{10}, c_{11}, c_{12}...$$

and `np.dot(V, c.flat)` and `legval2d(x, y, c)` will be the same up to roundoff. This equivalence is useful both for least squares fitting and for the evaluation of a large number of 2-D Legendre series of the same degrees and sample points.

**Parameters**

**x, y**

[array_like] Arrays of point coordinates, all of the same shape. The dtypes will be converted to either float64 or complex128 depending on whether any of the elements are complex. Scalars are converted to 1-D arrays.

**deg**

[list of ints] List of maximum degrees of the form [x_deg, y_deg].

**Returns**

**vander2d**

[ndarray] The shape of the returned matrix is `x.shape + (order,)`, where $order = (deg[0] + 1) * (deg[1] + 1)$. The dtype will be the same as the converted *x* and *y*.

**See also:**

*legvander*, *legvander3d*, *legval2d*, *legval3d*

## Notes

New in version 1.7.0.

`numpy.polynomial.legendre.`**`legvander3d`**`(`*x*, *y*, *z*, *deg*`)`

Pseudo-Vandermonde matrix of given degrees.

Returns the pseudo-Vandermonde matrix of degrees *deg* and sample points *(x, y, z)*. If *l, m, n* are the given degrees in *x, y, z*, then The pseudo-Vandermonde matrix is defined by

$$V[..., (m+1)(n+1)i + (n+1)j + k] = L_i(x) * L_j(y) * L_k(z),$$

where *0 <= i <= l*, *0 <= j <= m*, and *0 <= j <= n*. The leading indices of *V* index the points *(x, y, z)* and the last index encodes the degrees of the Legendre polynomials.

If `V = legvander3d(x, y, z, [xdeg, ydeg, zdeg])`, then the columns of *V* correspond to the elements of a 3-D coefficient array *c* of shape (xdeg + 1, ydeg + 1, zdeg + 1) in the order

$$c_{000}, c_{001}, c_{002}, ..., c_{010}, c_{011}, c_{012}, ...$$

and `np.dot(V, c.flat)` and `legval3d(x, y, z, c)` will be the same up to roundoff. This equivalence is useful both for least squares fitting and for the evaluation of a large number of 3-D Legendre series of the same degrees and sample points.

**Parameters**

**x, y, z**

[array_like] Arrays of point coordinates, all of the same shape. The dtypes will be converted to either float64 or complex128 depending on whether any of the elements are complex. Scalars are converted to 1-D arrays.

**deg**

[list of ints] List of maximum degrees of the form [x_deg, y_deg, z_deg].

**Returns**

**vander3d**

[ndarray] The shape of the returned matrix is `x.shape + (order,)`, where $order = (deg[0]+1)*(deg[1]+1)*(deg[2]+1)$. The dtype will be the same as the converted *x*, *y*, and *z*.

**See also:**

*legvander*, *legvander3d*, *legval2d*, *legval3d*

## Notes

New in version 1.7.0.

`numpy.polynomial.legendre.`**`leggauss`**`(`*deg*`)`

Gauss-Legendre quadrature.

Computes the sample points and weights for Gauss-Legendre quadrature. These sample points and weights will correctly integrate polynomials of degree $2*deg-1$ or less over the interval $[-1,1]$ with the weight function $f(x)=1$.

**Parameters**

**deg**

[int] Number of sample points and weights. It must be >= 1.

**Returns**

**x**

[ndarray] 1-D ndarray containing the sample points.

**y**

[ndarray] 1-D ndarray containing the weights.

## Notes

New in version 1.7.0.

The results have only been tested up to degree 100, higher degrees may be problematic. The weights are determined by using the fact that

$$w_k = c/(L'_n(x_k) * L_{n-1}(x_k))$$

where $c$ is a constant independent of $k$ and $x_k$ is the k'th root of $L_n$, and then scaling the results to get the right value when integrating 1.

`numpy.polynomial.legendre.`**`legweight`**`(`*x*`)`

Weight function of the Legendre polynomials.

The weight function is $1$ and the interval of integration is $[-1,1]$. The Legendre polynomials are orthogonal, but not normalized, with respect to this weight function.

**Parameters**

**x**

[array_like] Values at which the weight function will be computed.

**Returns**

**w**

[ndarray] The weight function at *x*.

## Notes

New in version 1.7.0.

`numpy.polynomial.legendre.`**`legcompanion`**(*c*)

Return the scaled companion matrix of c.

The basis polynomials are scaled so that the companion matrix is symmetric when *c* is an Legendre basis polynomial. This provides better eigenvalue estimates than the unscaled case and for basis polynomials the eigenvalues are guaranteed to be real if *`numpy.linalg.eigvalsh`* is used to obtain them.

**Parameters**

**c**

[array_like] 1-D array of Legendre series coefficients ordered from low to high degree.

**Returns**

**mat**

[ndarray] Scaled companion matrix of dimensions (deg, deg).

## Notes

New in version 1.7.0.

`numpy.polynomial.legendre.`**`legfit`**(*x*, *y*, *deg*, *rcond=None*, *full=False*, *w=None*)

Least squares fit of Legendre series to data.

Return the coefficients of a Legendre series of degree *deg* that is the least squares fit to the data values *y* given at points *x*. If *y* is 1-D the returned coefficients will also be 1-D. If *y* is 2-D multiple fits are done, one for each column of *y*, and the resulting coefficients are stored in the corresponding columns of a 2-D return. The fitted polynomial(s) are in the form

$$p(x) = c_0 + c_1 * L_1(x) + ... + c_n * L_n(x),$$

where *n* is *deg*.

**Parameters**

**x**

[array_like, shape (M,)] x-coordinates of the M sample points `(x[i], y[i])`.

**y**

[array_like, shape (M,) or (M, K)] y-coordinates of the sample points. Several data sets of sample points sharing the same x-coordinates can be fitted at once by passing in a 2D-array that contains one dataset per column.

**deg**

[int or 1-D array_like] Degree(s) of the fitting polynomials. If *deg* is a single integer all terms up to and including the *deg*'th term are included in the fit. For NumPy versions >= 1.11.0 a list of integers specifying the degrees of the terms to include may be used instead.

**rcond**

[float, optional] Relative condition number of the fit. Singular values smaller than this relative to the largest singular value will be ignored. The default value is len(x)*eps, where eps is the relative precision of the float type, about 2e-16 in most cases.

**full**

[bool, optional] Switch determining nature of return value. When it is False (the default) just the coefficients are returned, when True diagnostic information from the singular value decomposition is also returned.

**w**

[array_like, shape (*M*,), optional] Weights. If not None, the contribution of each point `(x[i],y[i])` to the fit is weighted by *w[i]*. Ideally the weights are chosen so that the errors of the products `w[i]*y[i]` all have the same variance. The default value is None.

New in version 1.5.0.

**Returns**

**coef**

[ndarray, shape (M,) or (M, K)] Legendre coefficients ordered from low to high. If *y* was 2-D, the coefficients for the data in column k of *y* are in column *k*. If *deg* is specified as a list, coefficients for terms not included in the fit are set equal to zero in the returned *coef*.

**[residuals, rank, singular_values, rcond]**

[list] These values are only returned if *full* = True

resid – sum of squared residuals of the least squares fit rank – the numerical rank of the scaled Vandermonde matrix sv – singular values of the scaled Vandermonde matrix rcond – value of *rcond*.

For more details, see *linalg.lstsq*.

**Warns**

**RankWarning**

The rank of the coefficient matrix in the least-squares fit is deficient. The warning is only raised if *full* = False. The warnings can be turned off by

```
>>> import warnings
>>> warnings.simplefilter('ignore', np.RankWarning)
```

**See also:**

`chebfit`, `polyfit`, `lagfit`, `hermfit`, `hermefit`

***`legval`***

Evaluates a Legendre series.

**`legvander`**

Vandermonde matrix of Legendre series.

**`legweight`**

Legendre weight function (= 1).

**`linalg.lstsq`**

Computes a least-squares fit from the matrix.

**`scipy.interpolate.UnivariateSpline`**

Computes spline fits.

### Notes

The solution is the coefficients of the Legendre series *p* that minimizes the sum of the weighted squared errors

$$E = \sum_j w_j^2 * |y_j - p(x_j)|^2,$$

where $w_j$ are the weights. This problem is solved by setting up as the (typically) overdetermined matrix equation

$$V(x) * c = w * y,$$

where *V* is the weighted pseudo Vandermonde matrix of *x*, *c* are the coefficients to be solved for, *w* are the weights, and *y* are the observed values. This equation is then solved using the singular value decomposition of *V*.

If some of the singular values of *V* are so small that they are neglected, then a *RankWarning* will be issued. This means that the coefficient values may be poorly determined. Using a lower order fit will usually get rid of the warning. The *rcond* parameter can also be set to a value smaller than its default, but the resulting fit may be spurious and have large contributions from roundoff error.

Fits using Legendre series are usually better conditioned than fits using power series, but much can depend on the distribution of the sample points and the smoothness of the data. If the quality of the fit is inadequate splines may be a good alternative.

### References

[1]

`numpy.polynomial.legendre.`**`legtrim`**`(`*c*, *tol=0*`)`

Remove "small" "trailing" coefficients from a polynomial.

"Small" means "small in absolute value" and is controlled by the parameter *tol*; "trailing" means highest order coefficient(s), e.g., in `[0, 1, 1, 0, 0]` (which represents `0 + x + x**2 + 0*x**3 + 0*x**4`) both the 3-rd and 4-th order coefficients would be "trimmed."

**Parameters**

**c**

[array_like] 1-d array of coefficients, ordered from lowest order to highest.

**tol**

[number, optional] Trailing (i.e., highest order) elements with absolute value less than or equal to *tol* (default value is zero) are removed.

**Returns**

**trimmed**

[ndarray] 1-d array with trailing zeros removed. If the resulting series would be empty, a series containing a single zero is returned.

**Raises**

**ValueError**

If *tol* < 0

**See also:**

`trimseq`

## Examples

```
>>> from numpy.polynomial import polyutils as pu
>>> pu.trimcoef((0,0,3,0,5,0,0))
array([0.,  0.,  3.,  0.,  5.])
>>> pu.trimcoef((0,0,1e-3,0,1e-5,0,0),1e-3) # item == tol is trimmed
array([0.])
>>> i = complex(0,1) # works for complex
>>> pu.trimcoef((3e-4,1e-3*(1-i),5e-4,2e-5*(1+i)), 1e-3)
array([0.0003+0.j   , 0.001 -0.001j])
```

`numpy.polynomial.legendre.`**`legline`**`(`*off*, *scl*`)`

Legendre series whose graph is a straight line.

**Parameters**

**off, scl**

[scalars] The specified line is given by `off + scl*x`.

**Returns**

**y**

[ndarray] This module's representation of the Legendre series for `off + scl*x`.

**See also:**

`polyline`, `chebline`

## Examples

```
>>> import numpy.polynomial.legendre as L
>>> L.legline(3,2)
array([3, 2])
>>> L.legval(-3, L.legline(3,2)) # should be -3
-3.0
```

`numpy.polynomial.legendre.`**`leg2poly`**`(`*c*`)`

Convert a Legendre series to a polynomial.

Convert an array representing the coefficients of a Legendre series, ordered from lowest degree to highest, to an array of the coefficients of the equivalent polynomial (relative to the "standard" basis) ordered from lowest to highest degree.

**Parameters**

**c**

[array_like] 1-D array containing the Legendre series coefficients, ordered from lowest order term to highest.

**Returns**

**pol**

[ndarray] 1-D array containing the coefficients of the equivalent polynomial (relative to the "standard" basis) ordered from lowest order term to highest.

**See also:**

*poly2leg*

## Notes

The easy way to do conversions between polynomial basis sets is to use the convert method of a class instance.

## Examples

```
>>> from numpy import polynomial as P
>>> c = P.Legendre(range(4))
>>> c
Legendre([0., 1., 2., 3.], domain=[-1,  1], window=[-1,  1])
>>> p = c.convert(kind=P.Polynomial)
>>> p
Polynomial([-1. , -3.5,  3. ,  7.5], domain=[-1.,  1.], window=[-1.,  1.])
>>> P.leg2poly(range(4))
array([-1. , -3.5,  3. ,  7.5])
```

`numpy.polynomial.legendre.`**`poly2leg`**`(`*pol*`)`

Convert a polynomial to a Legendre series.

Convert an array representing the coefficients of a polynomial (relative to the "standard" basis) ordered from lowest degree to highest, to an array of the coefficients of the equivalent Legendre series, ordered from lowest to highest degree.

**Parameters**

**pol**

[array_like] 1-D array containing the polynomial coefficients

**Returns**

**c**

[ndarray] 1-D array containing the coefficients of the equivalent Legendre series.

**See also:**

*leg2poly*

### Notes

The easy way to do conversions between polynomial basis sets is to use the convert method of a class instance.

### Examples

```
>>> from numpy import polynomial as P
>>> p = P.Polynomial(np.arange(4))
>>> p
Polynomial([0.,  1.,  2.,  3.], domain=[-1,  1], window=[-1,  1])
>>> c = P.Legendre(P.legendre.poly2leg(p.coef))
>>> c
Legendre([ 1.  ,  3.25,  1.  ,  0.75], domain=[-1,  1], window=[-1,  1]) # may␣
↪vary
```

## See also

numpy.polynomial

## Polyutils

Utility classes and functions for the polynomial modules.

This module provides: error and warning objects; a polynomial base class; and some routines used in both the *polynomial* and *chebyshev* modules.

### Error objects

| | |
|---|---|
| *PolyError* | Base class for errors in this module. |
| *PolyDomainError* | Issued by the generic Poly class when two domains don't match. |

**exception** `numpy.polynomial.polyutils.`**`PolyError`**

Base class for errors in this module.

**exception** `numpy.polynomial.polyutils.`**`PolyDomainError`**

Issued by the generic Poly class when two domains don't match.

This is raised when an binary operation is passed Poly objects with different domains.

### Warning objects

| | |
|---|---|
| *RankWarning* | Issued by chebfit when the design matrix is rank deficient. |

**exception** numpy.polynomial.polyutils.**RankWarning**

Issued by chebfit when the design matrix is rank deficient.

### Base class

| | |
|---|---|
| *PolyBase* | Base class for all polynomial types. |

**class** numpy.polynomial.polyutils.**PolyBase**

Base class for all polynomial types.

Deprecated in numpy 1.9.0, use the abstract ABCPolyBase class instead. Note that the latter requires a number of virtual functions to be implemented.

### Functions

| | |
|---|---|
| *as_series*(alist[, trim]) | Return argument as a list of 1-d arrays. |
| *trimseq*(seq) | Remove small Poly series coefficients. |
| *trimcoef*(c[, tol]) | Remove "small" "trailing" coefficients from a polynomial. |
| *getdomain*(x) | Return a domain suitable for given abscissae. |
| *mapdomain*(x, old, new) | Apply linear map to input points. |
| *mapparms*(old, new) | Linear map parameters between domains. |

numpy.polynomial.polyutils.**as_series**(*alist*, *trim=True*)

Return argument as a list of 1-d arrays.

The returned list contains array(s) of dtype double, complex double, or object. A 1-d argument of shape `(N,)` is parsed into `N` arrays of size one; a 2-d argument of shape `(M,N)` is parsed into `M` arrays of size `N` (i.e., is "parsed by row"); and a higher dimensional array raises a Value Error if it is not first reshaped into either a 1-d or 2-d array.

**Parameters**

**alist**

[array_like] A 1- or 2-d array_like

**trim**

[boolean, optional] When True, trailing zeros are removed from the inputs. When False, the inputs are passed through intact.

**Returns**

**[a1, a2,…]**

[list of 1-D arrays] A copy of the input data as a list of 1-d arrays.

**Raises**

**ValueError**

Raised when *as_series* cannot convert its input to 1-d arrays, or at least one of the resulting arrays is empty.

## Examples

```
>>> from numpy.polynomial import polyutils as pu
>>> a = np.arange(4)
>>> pu.as_series(a)
[array([0.]), array([1.]), array([2.]), array([3.])]
>>> b = np.arange(6).reshape((2,3))
>>> pu.as_series(b)
[array([0., 1., 2.]), array([3., 4., 5.])]
```

```
>>> pu.as_series((1, np.arange(3), np.arange(2, dtype=np.float16)))
[array([1.]), array([0., 1., 2.]), array([0., 1.])]
```

```
>>> pu.as_series([2, [1.1, 0.]])
[array([2.]), array([1.1])]
```

```
>>> pu.as_series([2, [1.1, 0.]], trim=False)
[array([2.]), array([1.1, 0. ])]
```

`numpy.polynomial.polyutils.`**`trimseq`**`(`*seq*`)`

Remove small Poly series coefficients.

**Parameters**

**seq**

[sequence] Sequence of Poly series coefficients. This routine fails for empty sequences.

**Returns**

**series**

[sequence] Subsequence with trailing zeros removed. If the resulting sequence would be empty, return the first element. The returned sequence may or may not be a view.

## Notes

Do not lose the type info if the sequence contains unknown objects.

`numpy.polynomial.polyutils.`**`trimcoef`**`(`*c*, *tol=0*`)`

Remove "small" "trailing" coefficients from a polynomial.

"Small" means "small in absolute value" and is controlled by the parameter *tol*; "trailing" means highest order coefficient(s), e.g., in `[0, 1, 1, 0, 0]` (which represents `0 + x + x**2 + 0*x**3 + 0*x**4`) both the 3-rd and 4-th order coefficients would be "trimmed."

**Parameters**

**c**

[array_like] 1-d array of coefficients, ordered from lowest order to highest.

**tol**

[number, optional] Trailing (i.e., highest order) elements with absolute value less than or equal to *tol* (default value is zero) are removed.

**Returns**

**trimmed**

[ndarray] 1-d array with trailing zeros removed. If the resulting series would be empty, a series containing a single zero is returned.

**Raises**

**ValueError**

If *tol* < 0

**See also:**

*trimseq*

### Examples

```
>>> from numpy.polynomial import polyutils as pu
>>> pu.trimcoef((0,0,3,0,5,0,0))
array([0.,  0.,  3.,  0.,  5.])
>>> pu.trimcoef((0,0,1e-3,0,1e-5,0,0),1e-3) # item == tol is trimmed
array([0.])
>>> i = complex(0,1) # works for complex
>>> pu.trimcoef((3e-4,1e-3*(1-i),5e-4,2e-5*(1+i)), 1e-3)
array([0.0003+0.j   , 0.001 -0.001j])
```

`numpy.polynomial.polyutils.`**`getdomain`**`(x)`

Return a domain suitable for given abscissae.

Find a domain suitable for a polynomial or Chebyshev series defined at the values supplied.

**Parameters**

**x**

[array_like] 1-d array of abscissae whose domain will be determined.

**Returns**

**domain**

[ndarray] 1-d array containing two values. If the inputs are complex, then the two returned points are the lower left and upper right corners of the smallest rectangle (aligned with the axes) in the complex plane containing the points *x*. If the inputs are real, then the two points are the ends of the smallest interval containing the points *x*.

**See also:**

*mapparms*, *mapdomain*

## Examples

```
>>> from numpy.polynomial import polyutils as pu
>>> points = np.arange(4)**2 - 5; points
array([-5, -4, -1,  4])
>>> pu.getdomain(points)
array([-5.,  4.])
>>> c = np.exp(complex(0,1)*np.pi*np.arange(12)/6) # unit circle
>>> pu.getdomain(c)
array([-1.-1.j,  1.+1.j])
```

`numpy.polynomial.polyutils.`**`mapdomain`**(*x*, *old*, *new*)

Apply linear map to input points.

The linear map `offset + scale*x` that maps the domain *old* to the domain *new* is applied to the points *x*.

**Parameters**

**x**

[array_like] Points to be mapped. If *x* is a subtype of ndarray the subtype will be preserved.

**old, new**

[array_like] The two domains that determine the map. Each must (successfully) convert to 1-d arrays containing precisely two values.

**Returns**

**x_out**

[ndarray] Array of points of the same shape as *x*, after application of the linear map between the two domains.

**See also:**

*getdomain*, *mapparms*

## Notes

Effectively, this implements:

$$x\_out = new[0] + m(x - old[0])$$

where

$$m = \frac{new[1] - new[0]}{old[1] - old[0]}$$

### Examples

```
>>> from numpy.polynomial import polyutils as pu
>>> old_domain = (-1,1)
>>> new_domain = (0,2*np.pi)
>>> x = np.linspace(-1,1,6); x
array([-1. , -0.6, -0.2,  0.2,  0.6,  1. ])
>>> x_out = pu.mapdomain(x, old_domain, new_domain); x_out
array([ 0.        ,  1.25663706,  2.51327412,  3.76991118,  5.02654825, # may vary
        6.28318531])
>>> x - pu.mapdomain(x_out, new_domain, old_domain)
array([0., 0., 0., 0., 0., 0.])
```

Also works for complex numbers (and thus can be used to map any line in the complex plane to any other line therein).

```
>>> i = complex(0,1)
>>> old = (-1 - i, 1 + i)
>>> new = (-1 + i, 1 - i)
>>> z = np.linspace(old[0], old[1], 6); z
array([-1. -1.j , -0.6-0.6j, -0.2-0.2j,  0.2+0.2j,  0.6+0.6j,  1. +1.j ])
>>> new_z = pu.mapdomain(z, old, new); new_z
array([-1.0+1.j , -0.6+0.6j, -0.2+0.2j,  0.2-0.2j,  0.6-0.6j,  1.0-1.j ]) # may␣
↪vary
```

`numpy.polynomial.polyutils.`**`mapparms`**`(`*old*, *new*`)`

Linear map parameters between domains.

Return the parameters of the linear map `offset + scale*x` that maps *old* to *new* such that `old[i] -> new[i]`, `i = 0, 1`.

**Parameters**

**old, new**

[array_like] Domains. Each domain must (successfully) convert to a 1-d array containing precisely two values.

**Returns**

**offset, scale**

[scalars] The map `L(x) = offset + scale*x` maps the first domain to the second.

**See also:**

*`getdomain`*, *`mapdomain`*

### Notes

Also works for complex numbers, and thus can be used to calculate the parameters required to map any line in the complex plane to any other line therein.

### Examples

```
>>> from numpy.polynomial import polyutils as pu
>>> pu.mapparms((-1,1),(-1,1))
(0.0, 1.0)
>>> pu.mapparms((1,-1),(-1,1))
(-0.0, -1.0)
>>> i = complex(0,1)
>>> pu.mapparms((-i,-1),(1,i))
((1+1j), (1-0j))
```

## Poly1d

### Basics

| | |
|---|---|
| *poly1d*(c_or_r[, r, variable]) | A one-dimensional polynomial class. |
| *polyval*(p, x) | Evaluate a polynomial at specific values. |
| *poly*(seq_of_zeros) | Find the coefficients of a polynomial with the given sequence of roots. |
| *roots*(p) | Return the roots of a polynomial with coefficients given in p. |

**class** numpy.**poly1d**(*c_or_r*, *r=False*, *variable=None*)

A one-dimensional polynomial class.

A convenience class, used to encapsulate "natural" operations on polynomials so that said operations may take on their customary form in code (see Examples).

**Parameters**

**c_or_r**

[array_like] The polynomial's coefficients, in decreasing powers, or if the value of the second parameter is True, the polynomial's roots (values where the polynomial evaluates to 0). For example, `poly1d([1, 2, 3])` returns an object that represents $x^2 + 2x + 3$, whereas `poly1d([1, 2, 3], True)` returns one that represents $(x-1)(x-2)(x-3) = x^3 - 6x^2 + 11x - 6$.

**r**

[bool, optional] If True, *c_or_r* specifies the polynomial's roots; the default is False.

**variable**

[str, optional] Changes the variable used when printing *p* from *x* to `variable` (see Examples).

### Examples

Construct the polynomial $x^2 + 2x + 3$:

```
>>> p = np.poly1d([1, 2, 3])
>>> print(np.poly1d(p))
   2
1 x + 2 x + 3
```

Evaluate the polynomial at $x = 0.5$:

```
>>> p(0.5)
4.25
```

Find the roots:

```
>>> p.r
array([-1.+1.41421356j, -1.-1.41421356j])
>>> p(p.r)
array([ -4.44089210e-16+0.j,  -4.44089210e-16+0.j]) # may vary
```

These numbers in the previous line represent (0, 0) to machine precision

Show the coefficients:

```
>>> p.c
array([1, 2, 3])
```

Display the order (the leading zero-coefficients are removed):

```
>>> p.order
2
```

Show the coefficient of the k-th power in the polynomial (which is equivalent to `p.c[-(i+1)]`):

```
>>> p[1]
2
```

Polynomials can be added, subtracted, multiplied, and divided (returns quotient and remainder):

```
>>> p * p
poly1d([ 1,  4, 10, 12,  9])
```

```
>>> (p**3 + 4) / p
(poly1d([ 1.,  4., 10., 12.,  9.]), poly1d([4.]))
```

`asarray(p)` gives the coefficient array, so polynomials can be used in all functions that accept arrays:

```
>>> p**2 # square of polynomial
poly1d([ 1,  4, 10, 12,  9])
```

```
>>> np.square(p) # square of individual coefficients
array([1, 4, 9])
```

The variable used in the string representation of *p* can be modified, using the `variable` parameter:

```
>>> p = np.poly1d([1,2,3], variable='z')
>>> print(p)
   2
1 z + 2 z + 3
```

Construct a polynomial from its roots:

```
>>> np.poly1d([1, 2], True)
poly1d([ 1., -3.,  2.])
```

This is the same polynomial as obtained by:

```
>>> np.poly1d([1, -1]) * np.poly1d([1, -2])
poly1d([ 1, -3,  2])
```

**Attributes**

**`c`**

The polynomial coefficients

**`coef`**

The polynomial coefficients

**`coefficients`**

The polynomial coefficients

**`coeffs`**

The polynomial coefficients

**`o`**

The order or degree of the polynomial

**`order`**

The order or degree of the polynomial

**`r`**

The roots of the polynomial, where self(x) == 0

*`roots`*

The roots of the polynomial, where self(x) == 0

**`variable`**

The name of the polynomial variable

### Methods

| | |
|---|---|
| `__call__`(self, val) | Call self as a function. |
| `deriv`(self[, m]) | Return a derivative of this polynomial. |
| `integ`(self[, m, k]) | Return an antiderivative (indefinite integral) of this polynomial. |

method

`poly1d.`**`__call__`**(*self*, *val*)

Call self as a function.

method

`poly1d.`**`deriv`**(*self*, *m=1*)

Return a derivative of this polynomial.

Refer to `polyder` for full documentation.

**See also:**

**`polyder`**

equivalent function

method

`poly1d.`**`integ`**(*self*, *m=1*, *k=0*)

Return an antiderivative (indefinite integral) of this polynomial.

Refer to `polyint` for full documentation.

**See also:**

**`polyint`**

equivalent function

`numpy.`**`polyval`**(*p*, *x*)

Evaluate a polynomial at specific values.

If *p* is of length N, this function returns the value:

```
p[0]*x**(N-1) + p[1]*x**(N-2) + ... + p[N-2]*x + p[N-1]
```

If *x* is a sequence, then *p(x)* is returned for each element of *x*. If *x* is another polynomial then the composite polynomial *p(x(t))* is returned.

**Parameters**

**p**

[array_like or poly1d object] 1D array of polynomial coefficients (including coefficients equal to zero) from highest degree to the constant term, or an instance of poly1d.

**x**

[array_like or poly1d object] A number, an array of numbers, or an instance of poly1d, at which to evaluate *p*.

**Returns**

**values**

[ndarray or poly1d] If *x* is a poly1d instance, the result is the composition of the two polynomials, i.e., *x* is "substituted" in *p* and the simplified result is returned. In addition, the type of *x* - array_like or poly1d - governs the type of the output: *x* array_like => *values* array_like, *x* a poly1d object => *values* is also.

**See also:**

**`poly1d`**

A polynomial class.

### Notes

Horner's scheme [1] is used to evaluate the polynomial. Even so, for polynomials of high degree the values may be inaccurate due to rounding errors. Use carefully.

If *x* is a subtype of `ndarray` the return value will be of the same type.

### References

[1]

### Examples

```
>>> np.polyval([3,0,1], 5)  # 3 * 5**2 + 0 * 5**1 + 1
76
>>> np.polyval([3,0,1], np.poly1d(5))
poly1d([76.])
>>> np.polyval(np.poly1d([3,0,1]), 5)
76
>>> np.polyval(np.poly1d([3,0,1]), np.poly1d(5))
poly1d([76.])
```

`numpy.`**`poly`**(*seq_of_zeros*)

Find the coefficients of a polynomial with the given sequence of roots.

Returns the coefficients of the polynomial whose leading coefficient is one for the given sequence of zeros (multiple roots must be included in the sequence as many times as their multiplicity; see Examples). A square matrix (or array, which will be treated as a matrix) can also be given, in which case the coefficients of the characteristic polynomial of the matrix are returned.

**Parameters**

**seq_of_zeros**

[array_like, shape (N,) or (N, N)] A sequence of polynomial roots, or a square array or matrix object.

**Returns**

**c**

[ndarray] 1D array of polynomial coefficients from highest to lowest degree:

`c[0] * x**(N) + c[1] * x**(N-1) + ... + c[N-1] * x + c[N]` where c[0] always equals 1.

**Raises**

**ValueError**

If input is the wrong shape (the input must be a 1-D or square 2-D array).

**See also:**

***polyval***

Compute polynomial values.

***roots***

Return the roots of a polynomial.

***polyfit***

Least squares polynomial fit.

***poly1d***

A one-dimensional polynomial class.

## Notes

Specifying the roots of a polynomial still leaves one degree of freedom, typically represented by an undetermined leading coefficient. [1] In the case of this function, that coefficient - the first one in the returned array - is always taken as one. (If for some reason you have one other point, the only automatic way presently to leverage that information is to use `polyfit`.)

The characteristic polynomial, $p_a(t)$, of an $n$-by-$n$ matrix **A** is given by

$$p_a(t) = \mathrm{det}(t\,\mathbf{I} - \mathbf{A}),$$

where **I** is the $n$-by-$n$ identity matrix. [2]

## References

[1], [2]

## Examples

Given a sequence of a polynomial's zeros:

```
>>> np.poly((0, 0, 0)) # Multiple root example
array([1., 0., 0., 0.])
```

The line above represents z**3 + 0*z**2 + 0*z + 0.

```
>>> np.poly((-1./2, 0, 1./2))
array([ 1.  ,  0.  , -0.25,  0.  ])
```

The line above represents z**3 - z/4

```
>>> np.poly((np.random.random(1)[0], 0, np.random.random(1)[0]))
array([ 1.        , -0.77086955,  0.08618131,  0.        ]) # random
```

Given a square array object:

```
>>> P = np.array([[0, 1./3], [-1./2, 0]])
>>> np.poly(P)
array([1.        , 0.        , 0.16666667])
```

Note how in all cases the leading coefficient is always 1.

`numpy.`**`roots`**(*p*)

Return the roots of a polynomial with coefficients given in p.

The values in the rank-1 array *p* are coefficients of a polynomial. If the length of *p* is n+1 then the polynomial is described by:

```
p[0] * x**n + p[1] * x**(n-1) + ... + p[n-1]*x + p[n]
```

**Parameters**

**p**

[array_like] Rank-1 array of polynomial coefficients.

**Returns**

**out**

[ndarray] An array containing the roots of the polynomial.

**Raises**

**ValueError**

When *p* cannot be converted to a rank-1 array.

**See also:**

**`poly`**

Find the coefficients of a polynomial with a given sequence of roots.

**`polyval`**

Compute polynomial values.

**`polyfit`**

Least squares polynomial fit.

**`poly1d`**

A one-dimensional polynomial class.

### Notes

The algorithm relies on computing the eigenvalues of the companion matrix [1].

### References

[1]

### Examples

```
>>> coeff = [3.2, 2, 1]
>>> np.roots(coeff)
array([-0.3125+0.46351241j, -0.3125-0.46351241j])
```

## Fitting

| | |
|---|---|
| `polyfit`(x, y, deg[, rcond, full, w, cov]) | Least squares polynomial fit. |

`numpy.`**`polyfit`**(*x*, *y*, *deg*, *rcond=None*, *full=False*, *w=None*, *cov=False*)

Least squares polynomial fit.

Fit a polynomial `p(x) = p[0] * x**deg + ... + p[deg]` of degree *deg* to points *(x, y)*. Returns a vector of coefficients *p* that minimises the squared error in the order *deg*, *deg-1*, … *0*.

The `Polynomial.fit` class method is recommended for new code as it is more stable numerically. See the documentation of the method for more information.

**Parameters**

**x**

[array_like, shape (M,)] x-coordinates of the M sample points `(x[i], y[i])`.

**y**

[array_like, shape (M,) or (M, K)] y-coordinates of the sample points. Several data sets of sample points sharing the same x-coordinates can be fitted at once by passing in a 2D-array that contains one dataset per column.

**deg**

[int] Degree of the fitting polynomial

**rcond**

[float, optional] Relative condition number of the fit. Singular values smaller than this relative to the largest singular value will be ignored. The default value is len(x)*eps, where eps is the relative precision of the float type, about 2e-16 in most cases.

**full**

[bool, optional] Switch determining nature of return value. When it is False (the default) just the coefficients are returned, when True diagnostic information from the singular value decomposition is also returned.

**w**

[array_like, shape (M,), optional] Weights to apply to the y-coordinates of the sample points. For gaussian uncertainties, use 1/sigma (not 1/sigma**2).

**cov**

[bool or str, optional] If given and not *False*, return not just the estimate but also its covariance matrix. By default, the covariance are scaled by chi2/sqrt(N-dof), i.e., the weights are presumed to be unreliable except in a relative sense and everything is scaled such that the reduced chi2 is unity. This scaling is omitted if `cov='unscaled'`, as is relevant for the case that the weights are 1/sigma**2, with sigma known to be a reliable estimate of the uncertainty.

**Returns**

**p**

[ndarray, shape (deg + 1,) or (deg + 1, K)] Polynomial coefficients, highest power first. If *y* was 2-D, the coefficients for *k*-th data set are in `p[:,k]`.

**residuals, rank, singular_values, rcond**

Present only if *`full`* = True. Residuals is sum of squared residuals of the least-squares fit, the effective rank of the scaled Vandermonde coefficient matrix, its singular values, and the specified value of *rcond*. For more details, see *`linalg.lstsq`*.

**V**

[ndarray, shape (M,M) or (M,M,K)] Present only if *`full`* = False and *cov'=True. The covariance matrix of the polynomial coefficient estimates. The diagonal of this matrix are the variance estimates for each coefficient. If y is a 2-D array, then the covariance matrix for the 'k*-th data set are in `V[:,:,k]`

**Warns**

**RankWarning**

The rank of the coefficient matrix in the least-squares fit is deficient. The warning is only raised if *`full`* = False.

The warnings can be turned off by

```
>>> import warnings
>>> warnings.simplefilter('ignore', np.RankWarning)
```

**See also:**

***`polyval`***

Compute polynomial values.

***`linalg.lstsq`***

Computes a least-squares fit.

**`scipy.interpolate.UnivariateSpline`**

Computes spline fits.

### Notes

The solution minimizes the squared error

$$E = \sum_{j=0}^{k} |p(x_j) - y_j|^2$$

in the equations:

```
x[0]**n * p[0] + ... + x[0] * p[n-1] + p[n] = y[0]
x[1]**n * p[0] + ... + x[1] * p[n-1] + p[n] = y[1]
...
x[k]**n * p[0] + ... + x[k] * p[n-1] + p[n] = y[k]
```

The coefficient matrix of the coefficients *p* is a Vandermonde matrix.

`polyfit` issues a `RankWarning` when the least-squares fit is badly conditioned. This implies that the best fit is not well-defined due to numerical error. The results may be improved by lowering the polynomial degree or by replacing *x* by *x* - *x*.mean(). The *rcond* parameter can also be set to a value smaller than its default, but the resulting fit may be spurious: including contributions from the small singular values can add numerical noise to the result.

Note that fitting polynomial coefficients is inherently badly conditioned when the degree of the polynomial is large or the interval of sample points is badly centered. The quality of the fit should always be checked in these cases. When polynomial fits are not satisfactory, splines may be a good alternative.

### References

[1], [2]

### Examples

```
>>> import warnings
>>> x = np.array([0.0, 1.0, 2.0, 3.0,  4.0,  5.0])
>>> y = np.array([0.0, 0.8, 0.9, 0.1, -0.8, -1.0])
>>> z = np.polyfit(x, y, 3)
>>> z
array([ 0.08703704, -0.81349206,  1.69312169, -0.03968254]) # may vary
```

It is convenient to use `poly1d` objects for dealing with polynomials:

```
>>> p = np.poly1d(z)
>>> p(0.5)
0.6143849206349179 # may vary
>>> p(3.5)
-0.34732142857143039 # may vary
>>> p(10)
22.579365079365115 # may vary
```

High-order polynomials may oscillate wildly:

```
>>> with warnings.catch_warnings():
...     warnings.simplefilter('ignore', np.RankWarning)
...     p30 = np.poly1d(np.polyfit(x, y, 30))
...
```

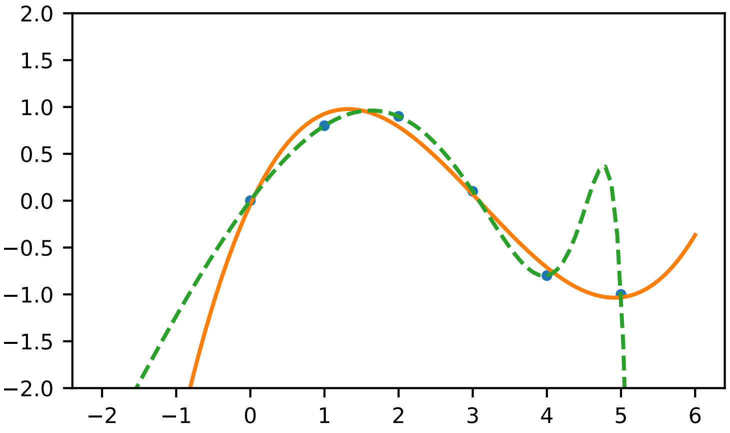
(continues on next page)

(continued from previous page)

```
>>> p30(4)
-0.80000000000000204 # may vary
>>> p30(5)
-0.99999999999999445 # may vary
>>> p30(4.5)
-0.10547061179440398 # may vary
```

Illustration:

```
>>> import matplotlib.pyplot as plt
>>> xp = np.linspace(-2, 6, 100)
>>> _ = plt.plot(x, y, '.', xp, p(xp), '-', xp, p30(xp), '--')
>>> plt.ylim(-2,2)
(-2, 2)
>>> plt.show()
```



## Calculus

| | |
|---|---|
| `polyder`(p[, m]) | Return the derivative of the specified order of a polynomial. |
| `polyint`(p[, m, k]) | Return an antiderivative (indefinite integral) of a polynomial. |

`numpy.`**`polyder`**(*p*, *m=1*)

Return the derivative of the specified order of a polynomial.

**Parameters**

**p**

[poly1d or sequence] Polynomial to differentiate. A sequence is interpreted as polynomial coefficients, see `poly1d`.

**m**

[int, optional] Order of differentiation (default: 1)

**Returns**

**der**

[poly1d] A new polynomial representing the derivative.

**See also:**

**`polyint`**

Anti-derivative of a polynomial.

**`poly1d`**

Class for one-dimensional polynomials.

## Examples

The derivative of the polynomial $x^3 + x^2 + x^1 + 1$ is:

```
>>> p = np.poly1d([1,1,1,1])
>>> p2 = np.polyder(p)
>>> p2
poly1d([3, 2, 1])
```

which evaluates to:

```
>>> p2(2.)
17.0
```

We can verify this, approximating the derivative with `(f(x + h) - f(x))/h`:

```
>>> (p(2. + 0.001) - p(2.)) / 0.001
17.007000999997857
```

The fourth-order derivative of a 3rd-order polynomial is zero:

```
>>> np.polyder(p, 2)
poly1d([6, 2])
>>> np.polyder(p, 3)
poly1d([6])
>>> np.polyder(p, 4)
poly1d([0.])
```

`numpy.`**`polyint`**(*p*, *m=1*, *k=None*)

Return an antiderivative (indefinite integral) of a polynomial.

The returned order *m* antiderivative *P* of polynomial *p* satisfies $\frac{d^m}{dx^m}P(x) = p(x)$ and is defined up to *m - 1* integration constants *k*. The constants determine the low-order polynomial part

$$\frac{k_{m-1}}{0!}x^0 + \ldots + \frac{k_0}{(m-1)!}x^{m-1}$$

of *P* so that $P^{(j)}(0) = k_{m-j-1}$.

**Parameters**

**p**

[array_like or poly1d] Polynomial to integrate. A sequence is interpreted as polynomial coefficients, see `poly1d`.

**m**

[int, optional] Order of the antiderivative. (Default: 1)

**k**

[list of *m* scalars or scalar, optional] Integration constants. They are given in the order of integration: those corresponding to highest-order terms come first.

If `None` (default), all constants are assumed to be zero. If *m = 1*, a single scalar can be given instead of a list.

**See also:**

**`polyder`**

derivative of a polynomial

**`poly1d.integ`**

equivalent method

## Examples

The defining property of the antiderivative:

```
>>> p = np.poly1d([1,1,1])
>>> P = np.polyint(p)
>>> P
 poly1d([ 0.33333333,  0.5       ,  1.        ,  0.        ]) # may vary
>>> np.polyder(P) == p
True
```

The integration constants default to zero, but can be specified:

```
>>> P = np.polyint(p, 3)
>>> P(0)
0.0
>>> np.polyder(P)(0)
0.0
>>> np.polyder(P, 2)(0)
0.0
>>> P = np.polyint(p, 3, k=[6,5,3])
>>> P
poly1d([ 0.01666667,  0.04166667,  0.16666667,  3. ,  5. ,  3. ]) # may vary
```

Note that 3 = 6 / 2!, and that the constants are given in the order of integrations. Constant of the highest-order polynomial term comes first:

```
>>> np.polyder(P, 2)(0)
6.0
>>> np.polyder(P, 1)(0)
5.0
>>> P(0)
3.0
```

### Arithmetic

| | |
|---|---|
| `polyadd`(a1, a2) | Find the sum of two polynomials. |
| `polydiv`(u, v) | Returns the quotient and remainder of polynomial division. |
| `polymul`(a1, a2) | Find the product of two polynomials. |
| `polysub`(a1, a2) | Difference (subtraction) of two polynomials. |

`numpy.`**`polyadd`**`(`*a1*, *a2*`)`

Find the sum of two polynomials.

Returns the polynomial resulting from the sum of two input polynomials. Each input must be either a poly1d object or a 1D sequence of polynomial coefficients, from highest to lowest degree.

**Parameters**

**a1, a2**

[array_like or poly1d object] Input polynomials.

**Returns**

**out**

[ndarray or poly1d object] The sum of the inputs. If either input is a poly1d object, then the output is also a poly1d object. Otherwise, it is a 1D array of polynomial coefficients from highest to lowest degree.

**See also:**

**`poly1d`**

A one-dimensional polynomial class.

`poly`, `polyadd`, `polyder`, `polydiv`, `polyfit`, `polyint`, `polysub`, `polyval`

#### Examples

```
>>> np.polyadd([1, 2], [9, 5, 4])
array([9, 6, 6])
```

Using poly1d objects:

```
>>> p1 = np.poly1d([1, 2])
>>> p2 = np.poly1d([9, 5, 4])
>>> print(p1)
1 x + 2
>>> print(p2)
   2
9 x + 5 x + 4
>>> print(np.polyadd(p1, p2))
   2
9 x + 6 x + 6
```

`numpy.`**`polydiv`**(*u*, *v*)

Returns the quotient and remainder of polynomial division.

The input arrays are the coefficients (including any coefficients equal to zero) of the "numerator" (dividend) and "denominator" (divisor) polynomials, respectively.

**Parameters**

**u**

[array_like or poly1d] Dividend polynomial's coefficients.

**v**

[array_like or poly1d] Divisor polynomial's coefficients.

**Returns**

**q**

[ndarray] Coefficients, including those equal to zero, of the quotient.

**r**

[ndarray] Coefficients, including those equal to zero, of the remainder.

**See also:**

*`poly`*, *`polyadd`*, *`polyder`*, *`polydiv`*, *`polyfit`*, *`polyint`*, *`polymul`*, *`polysub`*, *`polyval`*

## Notes

Both *u* and *v* must be 0-d or 1-d (ndim = 0 or 1), but *u.ndim* need not equal *v.ndim*. In other words, all four possible combinations - `u.ndim = v.ndim = 0`, `u.ndim = v.ndim = 1`, `u.ndim = 1, v.ndim = 0`, and `u.ndim = 0, v.ndim = 1` - work.

## Examples

$$\frac{3x^2 + 5x + 2}{2x + 1} = 1.5x + 1.75, remainder 0.25$$

```
>>> x = np.array([3.0, 5.0, 2.0])
>>> y = np.array([2.0, 1.0])
>>> np.polydiv(x, y)
(array([1.5 , 1.75]), array([0.25]))
```

`numpy.`**`polymul`**(*a1*, *a2*)

Find the product of two polynomials.

Finds the polynomial resulting from the multiplication of the two input polynomials. Each input must be either a poly1d object or a 1D sequence of polynomial coefficients, from highest to lowest degree.

**Parameters**

**a1, a2**

[array_like or poly1d object] Input polynomials.

**Returns**

**out**

[ndarray or poly1d object] The polynomial resulting from the multiplication of the inputs. If either inputs is a poly1d object, then the output is also a poly1d object. Otherwise, it is a 1D array of polynomial coefficients from highest to lowest degree.

**See also:**

**`poly1d`**

A one-dimensional polynomial class.

`poly`, `polyadd`, `polyder`, `polydiv`, `polyfit`, `polyint`, `polysub`, `polyval`

**`convolve`**

Array convolution. Same output as polymul, but has parameter for overlap mode.

### Examples

```
>>> np.polymul([1, 2, 3], [9, 5, 1])
array([ 9, 23, 38, 17,  3])
```

Using poly1d objects:

```
>>> p1 = np.poly1d([1, 2, 3])
>>> p2 = np.poly1d([9, 5, 1])
>>> print(p1)
   2
1 x + 2 x + 3
>>> print(p2)
   2
9 x + 5 x + 1
>>> print(np.polymul(p1, p2))
   4      3      2
9 x + 23 x + 38 x + 17 x + 3
```

`numpy.`**`polysub`**`(`*a1*, *a2*`)`

Difference (subtraction) of two polynomials.

Given two polynomials *a1* and *a2*, returns `a1 - a2`. *a1* and *a2* can be either array_like sequences of the polynomials' coefficients (including coefficients equal to zero), or `poly1d` objects.

**Parameters**

**a1, a2**

[array_like or poly1d] Minuend and subtrahend polynomials, respectively.

**Returns**

**out**

[ndarray or poly1d] Array or `poly1d` object of the difference polynomial's coefficients.

**See also:**

`polyval`, `polydiv`, `polymul`, `polyadd`

### Examples

$$(2x^2 + 10x - 2) - (3x^2 + 10x - 4) = (-x^2 + 2)$$

```
>>> np.polysub([2, 10, -2], [3, 10, -4])
array([-1,  0,  2])
```

### Warnings

| | |
|---|---|
| *RankWarning* | Issued by *polyfit* when the Vandermonde matrix is rank deficient. |

**exception** numpy.**RankWarning**

Issued by *polyfit* when the Vandermonde matrix is rank deficient.

For more information, a way to suppress the warning, and an example of *RankWarning* being issued, see *polyfit*.

## 4.24 Random sampling (`numpy.random`)

Numpy's random number routines produce pseudo random numbers using combinations of a *BitGenerator* to create sequences and a *Generator* to use those sequences to sample from different statistical distributions:

- BitGenerators: Objects that generate random numbers. These are typically unsigned integer words filled with sequences of either 32 or 64 random bits.
- Generators: Objects that transform sequences of random bits from a BitGenerator into sequences of numbers that follow a specific probability distribution (such as uniform, Normal or Binomial) within a specified interval.

Since Numpy version 1.17.0 the Generator can be initialized with a number of different BitGenerators. It exposes many different probability distributions. See NEP 19 for context on the updated random Numpy number routines. The legacy *RandomState* random number routines are still available, but limited to a single BitGenerator.

For convenience and backward compatibility, a single *RandomState* instance's methods are imported into the numpy.random namespace, see *Legacy Random Generation* for the complete list.

### 4.24.1 Quick Start

Call *default_rng* to get a new instance of a *Generator*, then call its methods to obtain samples from different distributions. By default, *Generator* uses bits provided by *PCG64* which has better statistical properties than the legacy *MT19937* used in *RandomState*.

```
# Do this
from numpy.random import default_rng
rng = default_rng()
vals = rng.standard_normal(10)
more_vals = rng.standard_normal(10)

# instead of this
from numpy import random
vals = random.standard_normal(10)
more_vals = random.standard_normal(10)
```

`Generator` can be used as a replacement for `RandomState`. Both class instances hold a internal `BitGenerator` instance to provide the bit stream, it is accessible as `gen.bit_generator`. Some long-overdue API cleanup means that legacy and compatibility methods have been removed from `Generator`

| `RandomState` | `Generator` | Notes |
|---|---|---|
| `random_sample, rand` | `random` | Compatible with `random.random` |
| `randint, random_integers` | `integers` | Add an `endpoint` kwarg |
| `tomaxint` | removed | Use `integers(0, np.iinfo(np.int_).max, endpoint=False)` |
| `seed` | removed | Use `SeedSequence.spawn` |

See *What's New or Different* for more information

Something like the following code can be used to support both `RandomState` and `Generator`, with the understanding that the interfaces are slightly different

```
try:
    rg_integers = rg.integers
except AttributeError:
    rg_integers = rg.randint
a = rg_integers(1000)
```

Seeds can be passed to any of the BitGenerators. The provided value is mixed via `SeedSequence` to spread a possible sequence of seeds across a wider range of initialization states for the BitGenerator. Here `PCG64` is used and is wrapped with a `Generator`.

```
from numpy.random import Generator, PCG64
rg = Generator(PCG64(12345))
rg.standard_normal()
```

## 4.24.2 Introduction

The new infrastructure takes a different approach to producing random numbers from the `RandomState` object. Random number generation is separated into two components, a bit generator and a random generator.

The `BitGenerator` has a limited set of responsibilities. It manages state and provides functions to produce random doubles and random unsigned 32- and 64-bit values.

The `random generator` takes the bit generator-provided stream and transforms them into more useful distributions, e.g., simulated normal random values. This structure allows alternative bit generators to be used with little code duplication.

The `Generator` is the user-facing object that is nearly identical to `RandomState`. The canonical method to initialize a generator passes a `PCG64` bit generator as the sole argument.

```
from numpy.random import default_rng
rg = default_rng(12345)
rg.random()
```

One can also instantiate `Generator` directly with a `BitGenerator` instance. To use the older `MT19937` algorithm, one can instantiate it directly and pass it to `Generator`.

```
from numpy.random import Generator, MT19937
rg = Generator(MT19937(12345))
rg.random()
```

### What's New or Different

**Warning:** The Box-Muller method used to produce NumPy's normals is no longer available in `Generator`. It is not possible to reproduce the exact random values using Generator for the normal distribution or any other distribution that relies on the normal such as the `RandomState.gamma` or `RandomState.standard_t`. If you require bitwise backward compatible streams, use `RandomState`.

- The Generator's normal, exponential and gamma functions use 256-step Ziggurat methods which are 2-10 times faster than NumPy's Box-Muller or inverse CDF implementations.
- Optional `dtype` argument that accepts `np.float32` or `np.float64` to produce either single or double prevision uniform random variables for select distributions
- Optional `out` argument that allows existing arrays to be filled for select distributions
- All BitGenerators can produce doubles, uint64s and uint32s via CTypes (`PCG64.ctypes`) and CFFI (`PCG64.cffi`). This allows the bit generators to be used in numba.
- The bit generators can be used in downstream projects via *Cython*.
- `Generator.integers` is now the canonical way to generate integer random numbers from a discrete uniform distribution. The `rand` and `randn` methods are only available through the legacy `RandomState`. The `endpoint` keyword can be used to specify open or closed intervals. This replaces both `randint` and the deprecated `random_integers`.
- `Generator.random` is now the canonical way to generate floating-point random numbers, which replaces `RandomState.random_sample`, *RandomState.sample*, and *RandomState.ranf*. This is consistent with Python's `random.random`.
- All BitGenerators in numpy use `SeedSequence` to convert seeds into initialized states.
- The addition of an `axis` keyword argument to methods such as `Generator.choice`, `Generator.permutation`, and `Generator.shuffle` improves support for sampling from and shuffling multidimensional arrays.

See *What's New or Different* for a complete list of improvements and differences from the traditional `Randomstate`.

### Parallel Generation

The included generators can be used in parallel, distributed applications in one of three ways:

- *SeedSequence spawning*
- *Independent Streams*
- *Jumping the BitGenerator state*

## 4.24.3 Concepts

### Random Generator

The *`Generator`* provides access to a wide range of distributions, and served as a replacement for *`RandomState`*. The main difference between the two is that `Generator` relies on an additional BitGenerator to manage state and generate the random bits, which are then transformed into random values from useful distributions. The default BitGenerator used by `Generator` is *`PCG64`*. The BitGenerator can be changed by passing an instantized BitGenerator to `Generator`.

`numpy.random.`**`default_rng`**`()`

Construct a new Generator with the default BitGenerator (PCG64).

**Parameters**

**seed**

[{None, int, array_like[ints], SeedSequence, BitGenerator, Generator}, optional] A seed to initialize the *`BitGenerator`*. If None, then fresh, unpredictable entropy will be pulled from the OS. If an `int` or `array_like[ints]` is passed, then it will be passed to *`SeedSequence`* to derive the initial *`BitGenerator`* state. One may also pass in a‘SeedSequence‘ instance Additionally, when passed a *`BitGenerator`*, it will be wrapped by *`Generator`*. If passed a *`Generator`*, it will be returned unaltered.

**Returns**

**Generator**

The initialized generator object.

### Notes

If `seed` is not a *`BitGenerator`* or a *`Generator`*, a new *`BitGenerator`* is instantiated. This function does not manage a default global instance.

**`class`** `numpy.random.`**`Generator`**`(`*bit_generator*`)`

Container for the BitGenerators.

`Generator` exposes a number of methods for generating random numbers drawn from a variety of probability distributions. In addition to the distribution-specific arguments, each method takes a keyword argument *size* that defaults to `None`. If *size* is `None`, then a single value is generated and returned. If *size* is an integer, then a 1-D array filled with generated values is returned. If *size* is a tuple, then an array with that shape is filled and returned.

The function *`numpy.random.default_rng`* will instantiate a *`Generator`* with numpy's default *`BitGenerator`*.

**No Compatibility Guarantee**

`Generator` does not provide a version compatibility guarantee. In particular, as better algorithms evolve the bit stream may change.

**Parameters**

**bit_generator**

[BitGenerator] BitGenerator to use as the core generator.

**See also:**

*default_rng*

Recommended constructor for `Generator`.

### Notes

The Python stdlib module `random` contains pseudo-random number generator with a number of methods that are similar to the ones available in `Generator`. It uses Mersenne Twister, and this bit generator can be accessed using `MT19937`. `Generator`, besides being NumPy-aware, has the advantage that it provides a much larger number of probability distributions to choose from.

### Examples

```
>>> from numpy.random import Generator, PCG64
>>> rg = Generator(PCG64())
>>> rg.standard_normal()
-0.203  # random
```

## Accessing the BitGenerator

| | |
|---|---|
| `bit_generator` | Gets the bit generator instance used by the generator |

attribute

`Generator.`**`bit_generator`**

Gets the bit generator instance used by the generator

**Returns**

**bit_generator**

[BitGenerator] The bit generator instance used by the generator

## Simple random data

| | |
|---|---|
| `integers`(low[, high, size, dtype, endpoint]) | Return random integers from *low* (inclusive) to *high* (exclusive), or if endpoint=True, *low* (inclusive) to *high* (inclusive). |
| `random`([size, dtype, out]) | Return random floats in the half-open interval [0.0, 1.0). |
| `choice`(a[, size, replace, p, axis, shuffle]) | Generates a random sample from a given 1-D array |
| `bytes`(length) | Return random bytes. |

method

`Generator.`**`integers`**(*low*, *high=None*, *size=None*, *dtype=np.int64*, *endpoint=False*)

Return random integers from *low* (inclusive) to *high* (exclusive), or if endpoint=True, *low* (inclusive) to *high* (inclusive). Replaces `RandomState.randint` (with endpoint=False) and `RandomState.random_integers` (with endpoint=True)

Return random integers from the "discrete uniform" distribution of the specified dtype. If *high* is None (the default), then results are from 0 to *low*.

**Parameters**

**low**

[int or array-like of ints] Lowest (signed) integers to be drawn from the distribution (unless `high=None`, in which case this parameter is 0 and this value is used for *high*).

**high**

[int or array-like of ints, optional] If provided, one above the largest (signed) integer to be drawn from the distribution (see above for behavior if `high=None`). If array-like, must contain integer values

**size**

[int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. Default is None, in which case a single value is returned.

**dtype**

[dtype, optional] Desired dtype of the result. Byteorder must be native. The default value is np.int64.

**endpoint**

[bool, optional] If true, sample from the interval [low, high] instead of the default [low, high) Defaults to False

**Returns**

**out**

[int or ndarray of ints] *size*-shaped array of random integers from the appropriate distribution, or a single such random int if *size* not provided.

## Notes

When using broadcasting with uint64 dtypes, the maximum value (2**64) cannot be represented as a standard integer type. The high array (or low if high is None) must have object dtype, e.g., array([2**64]).

## References

[1]

## Examples

```
>>> rng = np.random.default_rng()
>>> rng.integers(2, size=10)
array([1, 0, 0, 0, 1, 1, 0, 0, 1, 0])  # random
>>> rng.integers(1, size=10)
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
```

Generate a 2 x 4 array of ints between 0 and 4, inclusive:

```
>>> rng.integers(5, size=(2, 4))
array([[4, 0, 2, 1],
       [3, 2, 2, 0]])  # random
```

Generate a 1 x 3 array with 3 different upper bounds

```
>>> rng.integers(1, [3, 5, 10])
array([2, 2, 9])  # random
```

Generate a 1 by 3 array with 3 different lower bounds

```
>>> rng.integers([1, 5, 7], 10)
array([9, 8, 7])  # random
```

Generate a 2 by 4 array using broadcasting with dtype of uint8

```
>>> rng.integers([1, 3, 5, 7], [[10], [20]], dtype=np.uint8)
array([[ 8,  6,  9,  7],
       [ 1, 16,  9, 12]], dtype=uint8)  # random
```

method

`Generator.`**`random`**(*size=None*, *dtype=np.float64*, *out=None*)

Return random floats in the half-open interval [0.0, 1.0).

Results are from the "continuous uniform" distribution over the stated interval. To sample $Unif[a, b), b > a$ multiply the output of `random` by *(b-a)* and add *a*:

```
(b - a) * random() + a
```

**Parameters**

**size**

[int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. Default is None, in which case a single value is returned.

**dtype**

[dtype, optional] Desired dtype of the result, only *float64* and *float32* are supported. Byteorder must be native. The default value is np.float64.

**out**

[ndarray, optional] Alternative output array in which to place the result. If size is not None, it must have the same shape as the provided size and must match the type of the output values.

**Returns**

**out**

[float or ndarray of floats] Array of random floats of shape *size* (unless `size=None`, in which case a single float is returned).

### Examples

```
>>> rng = np.random.default_rng()
>>> rng.random()
0.47108547995356098 # random
>>> type(rng.random())
<class 'float'>
>>> rng.random((5,))
array([ 0.30220482,  0.86820401,  0.1654503 ,  0.11659149,  0.54323428]) # random
```

Three-by-two array of random numbers from [-5, 0):

```
>>> 5 * rng.random((3, 2)) - 5
array([[-3.99149989, -0.52338984], # random
       [-2.99091858, -0.79479508],
       [-1.23204345, -1.75224494]])
```

method

`Generator.`**`choice`**(*a*, *size=None*, *replace=True*, *p=None*, *axis=0*, *shuffle=True*)

Generates a random sample from a given 1-D array

**Parameters**

**a**

[{array_like, int}] If an ndarray, a random sample is generated from its elements. If an int, the random sample is generated from np.arange(a).

**size**

[{int, tuple[int]}, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn from the 1-d *a*. If *a* has more than one dimension, the *size* shape will be inserted into the *axis* dimension, so the output `ndim` will be `a.ndim - 1 + len(size)`. Default is None, in which case a single value is returned.

**replace**

[bool, optional] Whether the sample is with or without replacement

**p**

[1-D array_like, optional] The probabilities associated with each entry in a. If not given the sample assumes a uniform distribution over all entries in a.

**axis**

[int, optional] The axis along which the selection is performed. The default, 0, selects by row.

**shuffle**

[bool, optional] Whether the sample is shuffled when sampling without replacement. Default is True, False provides a speedup.

**Returns**

**samples**

[single item or ndarray] The generated random samples

**Raises**

**ValueError**

If a is an int and less than zero, if p is not 1-dimensional, if a is array-like with a size 0, if p is not a vector of probabilities, if a and p have different lengths, or if replace=False and the sample size is greater than the population size.

**See also:**

*integers*, *shuffle*, *permutation*

## Examples

Generate a uniform random sample from np.arange(5) of size 3:

```
>>> rng = np.random.default_rng()
>>> rng.choice(5, 3)
array([0, 3, 4]) # random
>>> #This is equivalent to rng.integers(0,5,3)
```

Generate a non-uniform random sample from np.arange(5) of size 3:

```
>>> rng.choice(5, 3, p=[0.1, 0, 0.3, 0.6, 0])
array([3, 3, 0]) # random
```

Generate a uniform random sample from np.arange(5) of size 3 without replacement:

```
>>> rng.choice(5, 3, replace=False)
array([3,1,0]) # random
>>> #This is equivalent to rng.permutation(np.arange(5))[:3]
```

Generate a non-uniform random sample from np.arange(5) of size 3 without replacement:

```
>>> rng.choice(5, 3, replace=False, p=[0.1, 0, 0.3, 0.6, 0])
array([2, 3, 0]) # random
```

Any of the above can be repeated with an arbitrary array-like instead of just integers. For instance:

```
>>> aa_milne_arr = ['pooh', 'rabbit', 'piglet', 'Christopher']
>>> rng.choice(aa_milne_arr, 5, p=[0.5, 0.1, 0.1, 0.3])
array(['pooh', 'pooh', 'pooh', 'Christopher', 'piglet'], # random
      dtype='<U11')
```

method

`Generator.`**`bytes`**`(`*length*`)`

Return random bytes.

**Parameters**

**length**

[int] Number of random bytes.

**Returns**

**out**

[str] String of length *length*.

### Examples

```
>>> np.random.default_rng().bytes(10)
' eh\x85\x022SZ\xbf\xa4' #random
```

## Permutations

| | |
|---|---|
| *shuffle*(x[, axis]) | Modify a sequence in-place by shuffling its contents. |
| *permutation*(x[, axis]) | Randomly permute a sequence, or return a permuted range. |

method

`Generator.`**`shuffle`**(*x*, *axis=0*)

Modify a sequence in-place by shuffling its contents.

The order of sub-arrays is changed but their contents remains the same.

**Parameters**

**x**

[array_like] The array or list to be shuffled.

**axis**

[int, optional] The axis which *x* is shuffled along. Default is 0. It is only supported on *ndarray* objects.

**Returns**

**None**

### Examples

```
>>> rng = np.random.default_rng()
>>> arr = np.arange(10)
>>> rng.shuffle(arr)
>>> arr
[1 7 5 2 9 4 3 6 0 8] # random
```

```
>>> arr = np.arange(9).reshape((3, 3))
>>> rng.shuffle(arr)
>>> arr
array([[3, 4, 5], # random
       [6, 7, 8],
       [0, 1, 2]])
```

```
>>> arr = np.arange(9).reshape((3, 3))
>>> rng.shuffle(arr, axis=1)
>>> arr
array([[2, 0, 1], # random
       [5, 3, 4],
       [8, 6, 7]])
```

method

`Generator.`**`permutation`**(*x*, *axis=0*)

Randomly permute a sequence, or return a permuted range.

**Parameters**

**x**

[int or array_like] If *x* is an integer, randomly permute `np.arange(x)`. If *x* is an array, make a copy and shuffle the elements randomly.

**axis**

[int, optional] The axis which *x* is shuffled along. Default is 0.

**Returns**

**out**

[ndarray] Permuted sequence or array range.

### Examples

```
>>> rng = np.random.default_rng()
>>> rng.permutation(10)
array([1, 7, 4, 3, 0, 9, 2, 5, 8, 6]) # random
```

```
>>> rng.permutation([1, 4, 9, 12, 15])
array([15,  1,  9,  4, 12]) # random
```

```
>>> arr = np.arange(9).reshape((3, 3))
>>> rng.permutation(arr)
array([[6, 7, 8], # random
       [0, 1, 2],
       [3, 4, 5]])
```

```
>>> rng.permutation("abc")
Traceback (most recent call last):
    ...
numpy.AxisError: axis 0 is out of bounds for array of dimension 0
```

```
>>> arr = np.arange(9).reshape((3, 3))
>>> rng.permutation(arr, axis=1)
array([[0, 2, 1], # random
       [3, 5, 4],
       [6, 8, 7]])
```

### Distributions

| | |
|---|---|
| `beta`(a, b[, size]) | Draw samples from a Beta distribution. |
| `binomial`(n, p[, size]) | Draw samples from a binomial distribution. |
| `chisquare`(df[, size]) | Draw samples from a chi-square distribution. |
| `dirichlet`(alpha[, size]) | Draw samples from the Dirichlet distribution. |
| `exponential`([scale, size]) | Draw samples from an exponential distribution. |
| `f`(dfnum, dfden[, size]) | Draw samples from an F distribution. |
| `gamma`(shape[, scale, size]) | Draw samples from a Gamma distribution. |
| `geometric`(p[, size]) | Draw samples from the geometric distribution. |
| `gumbel`([loc, scale, size]) | Draw samples from a Gumbel distribution. |
| `hypergeometric`(ngood, nbad, nsample[, size]) | Draw samples from a Hypergeometric distribution. |
| `laplace`([loc, scale, size]) | Draw samples from the Laplace or double exponential distribution with specified location (or mean) and scale (decay). |
| `logistic`([loc, scale, size]) | Draw samples from a logistic distribution. |
| `lognormal`([mean, sigma, size]) | Draw samples from a log-normal distribution. |
| `logseries`(p[, size]) | Draw samples from a logarithmic series distribution. |
| `multinomial`(n, pvals[, size]) | Draw samples from a multinomial distribution. |
| `multivariate_hypergeometric`(colors, nsample) | Generate variates from a multivariate hypergeometric distribution. |
| `multivariate_normal`(mean, cov[, size, …]) | Draw random samples from a multivariate normal distribution. |
| `negative_binomial`(n, p[, size]) | Draw samples from a negative binomial distribution. |
| `noncentral_chisquare`(df, nonc[, size]) | Draw samples from a noncentral chi-square distribution. |
| `noncentral_f`(dfnum, dfden, nonc[, size]) | Draw samples from the noncentral F distribution. |
| `normal`([loc, scale, size]) | Draw random samples from a normal (Gaussian) distribution. |
| `pareto`(a[, size]) | Draw samples from a Pareto II or Lomax distribution with specified shape. |
| `poisson`([lam, size]) | Draw samples from a Poisson distribution. |
| `power`(a[, size]) | Draws samples in [0, 1] from a power distribution with positive exponent a - 1. |
| `rayleigh`([scale, size]) | Draw samples from a Rayleigh distribution. |
| `standard_cauchy`([size]) | Draw samples from a standard Cauchy distribution with mode = 0. |
| `standard_exponential`([size, dtype, method, out]) | Draw samples from the standard exponential distribution. |
| `standard_gamma`(shape[, size, dtype, out]) | Draw samples from a standard Gamma distribution. |
| `standard_normal`([size, dtype, out]) | Draw samples from a standard Normal distribution (mean=0, stdev=1). |
| `standard_t`(df[, size]) | Draw samples from a standard Student's t distribution with *df* degrees of freedom. |
| `triangular`(left, mode, right[, size]) | Draw samples from the triangular distribution over the interval `[left, right]`. |
| `uniform`([low, high, size]) | Draw samples from a uniform distribution. |
| `vonmises`(mu, kappa[, size]) | Draw samples from a von Mises distribution. |
| `wald`(mean, scale[, size]) | Draw samples from a Wald, or inverse Gaussian, distribution. |
| `weibull`(a[, size]) | Draw samples from a Weibull distribution. |
| `zipf`(a[, size]) | Draw samples from a Zipf distribution. |

method

`Generator.`**`beta`**`(`*a*, *b*, *size=None*`)`

Draw samples from a Beta distribution.

The Beta distribution is a special case of the Dirichlet distribution, and is related to the Gamma distribution. It has the probability distribution function

$$f(x; a, b) = \frac{1}{B(\alpha, \beta)} x^{\alpha - 1} (1 - x)^{\beta - 1},$$

where the normalization, B, is the beta function,

$$B(\alpha, \beta) = \int_0^1 t^{\alpha - 1} (1 - t)^{\beta - 1} dt.$$

It is often seen in Bayesian inference and order statistics.

**Parameters**

**a**

[float or array_like of floats] Alpha, positive (>0).

**b**

[float or array_like of floats] Beta, positive (>0).

**size**

[int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `a` and `b` are both scalars. Otherwise, `np.broadcast(a, b).size` samples are drawn.

**Returns**

**out**

[ndarray or scalar] Drawn samples from the parameterized beta distribution.

method

`Generator.`**`binomial`**`(`*n*, *p*, *size=None*`)`

Draw samples from a binomial distribution.

Samples are drawn from a binomial distribution with specified parameters, n trials and p probability of success where n an integer >= 0 and p is in the interval [0,1]. (n may be input as a float, but it is truncated to an integer in use)

**Parameters**

**n**

[int or array_like of ints] Parameter of the distribution, >= 0. Floats are also accepted, but they will be truncated to integers.

**p**

[float or array_like of floats] Parameter of the distribution, >= 0 and <=1.

**size**

[int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `n` and `p` are both scalars. Otherwise, `np.broadcast(n, p).size` samples are drawn.

**Returns**

**out**

[ndarray or scalar] Drawn samples from the parameterized binomial distribution, where each sample is equal to the number of successes over the n trials.

**See also:**

**`scipy.stats.binom`**

probability density function, distribution or cumulative density function, etc.

## Notes

The probability density for the binomial distribution is

$$P(N) = \binom{n}{N}p^N(1-p)^{n-N},$$

where $n$ is the number of trials, $p$ is the probability of success, and $N$ is the number of successes.

When estimating the standard error of a proportion in a population by using a random sample, the normal distribution works well unless the product p*n <=5, where p = population proportion estimate, and n = number of samples, in which case the binomial distribution is used instead. For example, a sample of 15 people shows 4 who are left handed, and 11 who are right handed. Then p = 4/15 = 27%. 0.27*15 = 4, so the binomial distribution should be used in this case.

## References

[1], [2], [3], [4], [5]

## Examples

Draw samples from the distribution:

```
>>> rng = np.random.default_rng()
>>> n, p = 10, .5  # number of trials, probability of each trial
>>> s = rng.binomial(n, p, 1000)
# result of flipping a coin 10 times, tested 1000 times.
```

A real world example. A company drills 9 wild-cat oil exploration wells, each with an estimated probability of success of 0.1. All nine wells fail. What is the probability of that happening?

Let's do 20,000 trials of the model, and count the number that generate zero positive results.

```
>>> sum(rng.binomial(9, 0.1, 20000) == 0)/20000.
# answer = 0.38885, or 38%.
```

method

`Generator.`**`chisquare`**(*df*, *size=None*)

Draw samples from a chi-square distribution.

When *df* independent random variables, each with standard normal distributions (mean 0, variance 1), are squared and summed, the resulting distribution is chi-square (see Notes). This distribution is often used in hypothesis testing.

**Parameters**

**df**

[float or array_like of floats] Number of degrees of freedom, must be > 0.

**size**

[int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `df` is a scalar. Otherwise, `np.array(df).size` samples are drawn.

**Returns**

**out**

[ndarray or scalar] Drawn samples from the parameterized chi-square distribution.

**Raises**

**ValueError**

When *df* <= 0 or when an inappropriate *size* (e.g. `size=-1`) is given.

## Notes

The variable obtained by summing the squares of *df* independent, standard normally distributed random variables:

$$Q = \sum_{i=0}^{\mathtt{df}} X_i^2$$

is chi-square distributed, denoted

$$Q \sim \chi_k^2.$$

The probability density function of the chi-squared distribution is

$$p(x) = \frac{(1/2)^{k/2}}{\Gamma(k/2)} x^{k/2-1} e^{-x/2},$$

where $\Gamma$ is the gamma function,

$$\Gamma(x) = \int_0^{-\infty} t^{x-1} e^{-t} dt.$$

## References

[1]

## Examples

```
>>> np.random.default_rng().chisquare(2,4)
array([ 1.89920014,  9.00867716,  3.13710533,  5.62318272]) # random
```

method

`Generator.`**`dirichlet`**(*alpha*, *size=None*)

Draw samples from the Dirichlet distribution.

Draw *size* samples of dimension k from a Dirichlet distribution. A Dirichlet-distributed random variable can be seen as a multivariate generalization of a Beta distribution. The Dirichlet distribution is a conjugate prior of a multinomial distribution in Bayesian inference.

**Parameters**

**alpha**

[sequence of floats, length k] Parameter of the distribution (length `k` for sample of length `k`).

**size**

[int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n)`, then `m * n * k` samples are drawn. Default is None, in which case a vector of length `k` is returned.

**Returns**

**samples**

[ndarray,] The drawn samples, of shape `(size, k)`.

**Raises**

**ValueError**

If any value in `alpha` is less than or equal to zero

## Notes

The Dirichlet distribution is a distribution over vectors $x$ that fulfil the conditions $x_i > 0$ and $\sum_{i=1}^{k} x_i = 1$.

The probability density function $p$ of a Dirichlet-distributed random vector $X$ is proportional to

$$p(x) \propto \prod_{i=1}^{k} x_i^{\alpha_i - 1},$$

where $\alpha$ is a vector containing the positive concentration parameters.

The method uses the following property for computation: let $Y$ be a random vector which has components that follow a standard gamma distribution, then $X = \frac{1}{\sum_{i=1}^{k} Y_i} Y$ is Dirichlet-distributed

### References

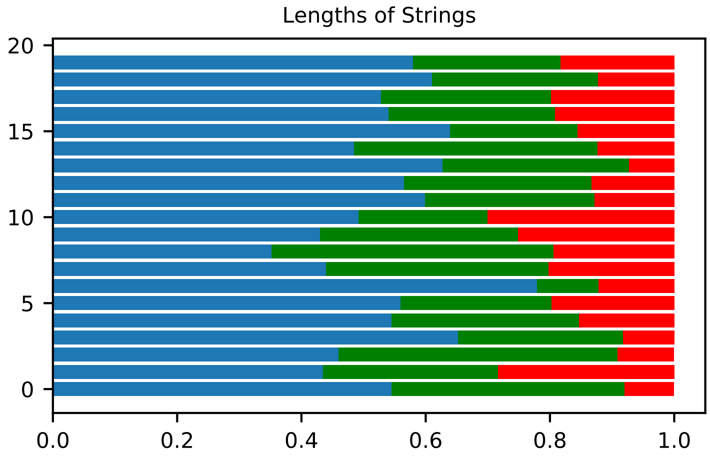
[1], [2]

### Examples

Taking an example cited in Wikipedia, this distribution can be used if one wanted to cut strings (each of initial length 1.0) into K pieces with different lengths, where each piece had, on average, a designated average length, but allowing some variation in the relative sizes of the pieces.

```
>>> s = np.random.default_rng().dirichlet((10, 5, 3), 20).transpose()
```

```
>>> import matplotlib.pyplot as plt
>>> plt.barh(range(20), s[0])
>>> plt.barh(range(20), s[1], left=s[0], color='g')
>>> plt.barh(range(20), s[2], left=s[0]+s[1], color='r')
>>> plt.title("Lengths of Strings")
```



method

`Generator.`**`exponential`**(*scale=1.0*, *size=None*)

Draw samples from an exponential distribution.

Its probability density function is

$$f(x; \frac{1}{\beta}) = \frac{1}{\beta} \exp(-\frac{x}{\beta}),$$

for `x > 0` and 0 elsewhere. $\beta$ is the scale parameter, which is the inverse of the rate parameter $\lambda = 1/\beta$. The rate parameter is an alternative, widely used parameterization of the exponential distribution [3].

The exponential distribution is a continuous analogue of the geometric distribution. It describes many common situations, such as the size of raindrops measured over many rainstorms [1], or the time between page requests to Wikipedia [2].

**Parameters**

**scale**

[float or array_like of floats] The scale parameter, $\beta = 1/\lambda$. Must be non-negative.

**size**

[int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `scale` is a scalar. Otherwise, `np.array(scale).size` samples are drawn.

**Returns**

**out**

[ndarray or scalar] Drawn samples from the parameterized exponential distribution.

### References

[1], [2], [3]

method

`Generator.`**`f`**`(`*dfnum*, *dfden*, *size=None*`)`

Draw samples from an F distribution.

Samples are drawn from an F distribution with specified parameters, *dfnum* (degrees of freedom in numerator) and *dfden* (degrees of freedom in denominator), where both parameters must be greater than zero.

The random variate of the F distribution (also known as the Fisher distribution) is a continuous probability distribution that arises in ANOVA tests, and is the ratio of two chi-square variates.

**Parameters**

**dfnum**

[float or array_like of floats] Degrees of freedom in numerator, must be > 0.

**dfden**

[float or array_like of float] Degrees of freedom in denominator, must be > 0.

**size**

[int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `dfnum` and `dfden` are both scalars. Otherwise, `np.broadcast(dfnum, dfden).size` samples are drawn.

**Returns**

**out**

[ndarray or scalar] Drawn samples from the parameterized Fisher distribution.

**See also:**

**`scipy.stats.f`**

probability density function, distribution or cumulative density function, etc.

### Notes

The F statistic is used to compare in-group variances to between-group variances. Calculating the distribution depends on the sampling, and so it is a function of the respective degrees of freedom in the problem. The variable *dfnum* is the number of samples minus one, the between-groups degrees of freedom, while *dfden* is the within-groups degrees of freedom, the sum of the number of samples in each group minus the number of groups.

### References

[1], [2]

### Examples

An example from Glantz[1], pp 47-40:

Two groups, children of diabetics (25 people) and children from people without diabetes (25 controls). Fasting blood glucose was measured, case group had a mean value of 86.1, controls had a mean value of 82.2. Standard deviations were 2.09 and 2.49 respectively. Are these data consistent with the null hypothesis that the parents diabetic status does not affect their children's blood glucose levels? Calculating the F statistic from the data gives a value of 36.01.

Draw samples from the distribution:

```
>>> dfnum = 1. # between group degrees of freedom
>>> dfden = 48. # within groups degrees of freedom
>>> s = np.random.default_rng().f(dfnum, dfden, 1000)
```

The lower bound for the top 1% of the samples is :

```
>>> np.sort(s)[-10]
7.61988120985 # random
```

So there is about a 1% chance that the F statistic will exceed 7.62, the measured value is 36, so the null hypothesis is rejected at the 1% level.

method

`Generator.`**`gamma`**(*shape*, *scale=1.0*, *size=None*)

Draw samples from a Gamma distribution.

Samples are drawn from a Gamma distribution with specified parameters, *shape* (sometimes designated "k") and *scale* (sometimes designated "theta"), where both parameters are > 0.

**Parameters**

**shape**

[float or array_like of floats] The shape of the gamma distribution. Must be non-negative.

**scale**

[float or array_like of floats, optional] The scale of the gamma distribution. Must be non-negative. Default is equal to 1.

**size**

[int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `shape` and `scale` are both scalars. Otherwise, `np.broadcast(shape, scale).size` samples are drawn.

**Returns**

**out**

[ndarray or scalar] Drawn samples from the parameterized gamma distribution.

**See also:**

**`scipy.stats.gamma`**

probability density function, distribution or cumulative density function, etc.

## Notes

The probability density for the Gamma distribution is

$$p(x) = x^{k-1}\frac{e^{-x/\theta}}{\theta^k\Gamma(k)},$$

where $k$ is the shape and $\theta$ the scale, and $\Gamma$ is the Gamma function.

The Gamma distribution is often used to model the times to failure of electronic components, and arises naturally in processes for which the waiting times between Poisson distributed events are relevant.

## References

[1], [2]

## Examples

Draw samples from the distribution:

```
>>> shape, scale = 2., 2.  # mean=4, std=2*sqrt(2)
>>> s = np.random.default_rng().gamma(shape, scale, 1000)
```

Display the histogram of the samples, along with the probability density function:

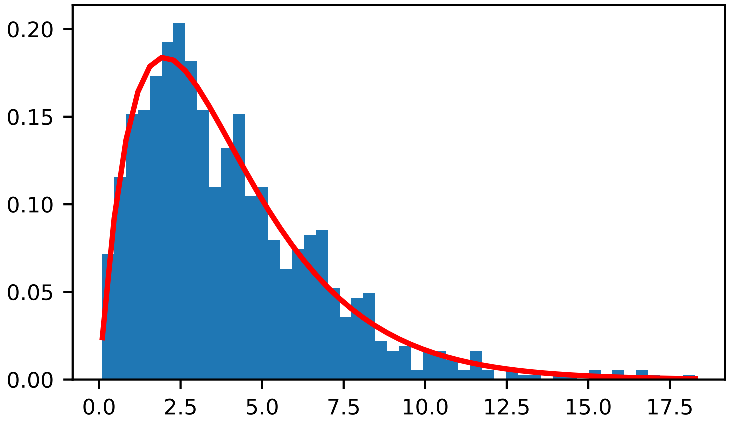
```
>>> import matplotlib.pyplot as plt
>>> import scipy.special as sps
>>> count, bins, ignored = plt.hist(s, 50, density=True)
>>> y = bins**(shape-1)*(np.exp(-bins/scale) /
...                      (sps.gamma(shape)*scale**shape))
>>> plt.plot(bins, y, linewidth=2, color='r')
>>> plt.show()
```

method

`Generator.`**`geometric`**(*p*, *size=None*)

Draw samples from the geometric distribution.

Bernoulli trials are experiments with one of two outcomes: success or failure (an example of such an experiment is flipping a coin). The geometric distribution models the number of trials that must be run in order to achieve success. It is therefore supported on the positive integers, `k = 1, 2, ...`.



The probability mass function of the geometric distribution is

$$f(k) = (1 - p)^{k-1} p$$

where $p$ is the probability of success of an individual trial.

**Parameters**

**p**

[float or array_like of floats] The probability of success of an individual trial.

**size**

[int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `p` is a scalar. Otherwise, `np.array(p).size` samples are drawn.

**Returns**

**out**

[ndarray or scalar] Drawn samples from the parameterized geometric distribution.

## Examples

Draw ten thousand values from the geometric distribution, with the probability of an individual success equal to 0.35:

```
>>> z = np.random.default_rng().geometric(p=0.35, size=10000)
```

How many trials succeeded after a single run?

```
>>> (z == 1).sum() / 10000.
0.34889999999999999 #random
```

method

`Generator.`**`gumbel`**(*loc=0.0*, *scale=1.0*, *size=None*)

Draw samples from a Gumbel distribution.

Draw samples from a Gumbel distribution with specified location and scale. For more information on the Gumbel distribution, see Notes and References below.

**Parameters**

**loc**

[float or array_like of floats, optional] The location of the mode of the distribution. Default is 0.

**scale**

[float or array_like of floats, optional] The scale parameter of the distribution. Default is 1. Must be non- negative.

**size**

[int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `loc` and `scale` are both scalars. Otherwise, `np.broadcast(loc, scale).size` samples are drawn.

**Returns**

**out**

[ndarray or scalar] Drawn samples from the parameterized Gumbel distribution.

**See also:**

`scipy.stats.gumbel_l`, `scipy.stats.gumbel_r`, `scipy.stats.genextreme`, *`weibull`*

## Notes

The Gumbel (or Smallest Extreme Value (SEV) or the Smallest Extreme Value Type I) distribution is one of a class of Generalized Extreme Value (GEV) distributions used in modeling extreme value problems. The Gumbel is a special case of the Extreme Value Type I distribution for maximums from distributions with "exponential-like" tails.

The probability density for the Gumbel distribution is

$$p(x) = \frac{e^{-(x - \mu)/ \beta}}{\beta} e^{ -e^{-(x - \mu)/ \beta}},$$

where $\mu$ is the mode, a location parameter, and $\beta$ is the scale parameter.

The Gumbel (named for German mathematician Emil Julius Gumbel) was used very early in the hydrology literature, for modeling the occurrence of flood events. It is also used for modeling maximum wind speed and rainfall rates. It is a "fat-tailed" distribution - the probability of an event in the tail of the distribution is larger than if one used a Gaussian, hence the surprisingly frequent occurrence of 100-year floods. Floods were initially modeled as a Gaussian process, which underestimated the frequency of extreme events.

It is one of a class of extreme value distributions, the Generalized Extreme Value (GEV) distributions, which also includes the Weibull and Frechet.

The function has a mean of $\mu + 0.57721\beta$ and a variance of $\frac{\pi^2}{6}\beta^2$.

### References

[1], [2]

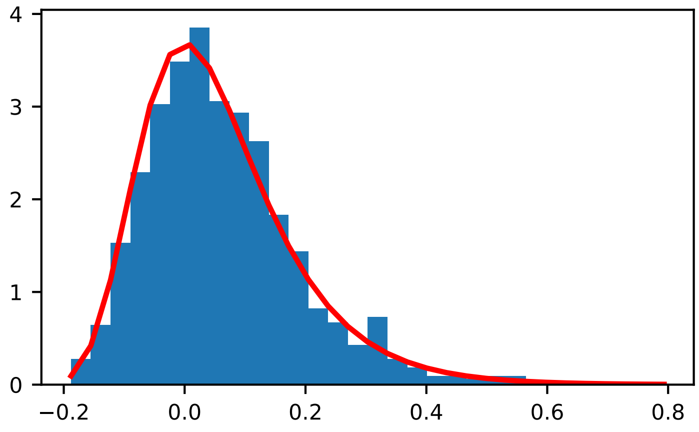
### Examples

Draw samples from the distribution:

```
>>> rng = np.random.default_rng()
>>> mu, beta = 0, 0.1 # location and scale
>>> s = rng.gumbel(mu, beta, 1000)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 30, density=True)
>>> plt.plot(bins, (1/beta)*np.exp(-(bins - mu)/beta)
...          * np.exp( -np.exp( -(bins - mu) /beta) ),
...          linewidth=2, color='r')
>>> plt.show()
```



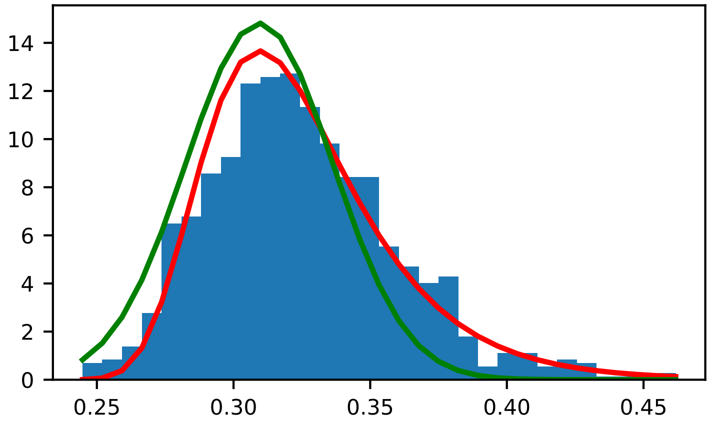
Show how an extreme value distribution can arise from a Gaussian process and compare to a Gaussian:

```
>>> means = []
>>> maxima = []
>>> for i in range(0,1000) :
...    a = rng.normal(mu, beta, 1000)
...    means.append(a.mean())
...    maxima.append(a.max())
>>> count, bins, ignored = plt.hist(maxima, 30, density=True)
>>> beta = np.std(maxima) * np.sqrt(6) / np.pi
>>> mu = np.mean(maxima) - 0.57721*beta
>>> plt.plot(bins, (1/beta)*np.exp(-(bins - mu)/beta)
...          * np.exp(-np.exp(-(bins - mu)/beta)),
...          linewidth=2, color='r')
```

(continues on next page)

(continued from previous page)

```
>>> plt.plot(bins, 1/(beta * np.sqrt(2 * np.pi))
...          * np.exp(-(bins - mu)**2 / (2 * beta**2)),
...          linewidth=2, color='g')
>>> plt.show()
```



method

`Generator.`**`hypergeometric`**(*ngood*, *nbad*, *nsample*, *size=None*)

Draw samples from a Hypergeometric distribution.

Samples are drawn from a hypergeometric distribution with specified parameters, *ngood* (ways to make a good selection), *nbad* (ways to make a bad selection), and *nsample* (number of items sampled, which is less than or equal to the sum `ngood + nbad`).

**Parameters**

**ngood**

[int or array_like of ints] Number of ways to make a good selection. Must be nonnegative and less than 10**9.

**nbad**

[int or array_like of ints] Number of ways to make a bad selection. Must be nonnegative and less than 10**9.

**nsample**

[int or array_like of ints] Number of items sampled. Must be nonnegative and less than `ngood + nbad`.

**size**

[int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if *ngood*, *nbad*, and *nsample* are all scalars. Otherwise, `np.broadcast(ngood, nbad, nsample).size` samples are drawn.

**Returns**

**out**

[ndarray or scalar] Drawn samples from the parameterized hypergeometric distribution. Each sample is the number of good items within a randomly selected subset of size *nsample* taken from a set of *ngood* good items and *nbad* bad items.

**See also:**

**`multivariate_hypergeometric`**

Draw samples from the multivariate hypergeometric distribution.

**`scipy.stats.hypergeom`**

probability density function, distribution or cumulative density function, etc.

## Notes

The probability density for the Hypergeometric distribution is

$$P(x) = \frac{\binom{g}{x}\binom{b}{n-x}}{\binom{g+b}{n}},$$

where $0 \le x \le n$ and $n - b \le x \le g$

for P(x) the probability of `x` good results in the drawn sample, g = *ngood*, b = *nbad*, and n = *nsample*.

Consider an urn with black and white marbles in it, *ngood* of them are black and *nbad* are white. If you draw *nsample* balls without replacement, then the hypergeometric distribution describes the distribution of black balls in the drawn sample.

Note that this distribution is very similar to the binomial distribution, except that in this case, samples are drawn without replacement, whereas in the Binomial case samples are drawn with replacement (or the sample space is infinite). As the sample space becomes large, this distribution approaches the binomial.

The arguments *ngood* and *nbad* each must be less than *10**9*. For extremely large arguments, the algorithm that is used to compute the samples [4] breaks down because of loss of precision in floating point calculations. For such large values, if *nsample* is not also large, the distribution can be approximated with the binomial distribution, *binomial(n=nsample, p=ngood/(ngood + nbad))*.

## References

[1], [2], [3], [4]

## Examples

Draw samples from the distribution:

```
>>> rng = np.random.default_rng()
>>> ngood, nbad, nsamp = 100, 2, 10
# number of good, number of bad, and number of samples
>>> s = rng.hypergeometric(ngood, nbad, nsamp, 1000)
>>> from matplotlib.pyplot import hist
>>> hist(s)
#   note that it is very unlikely to grab both bad items
```

Suppose you have an urn with 15 white and 15 black marbles. If you pull 15 marbles at random, how likely is it that 12 or more of them are one color?

```
>>> s = rng.hypergeometric(15, 15, 15, 100000)
>>> sum(s>=12)/100000. + sum(s<=3)/100000.
#   answer = 0.003 ... pretty unlikely!
```

method

`Generator.`**`laplace`**(*loc=0.0*, *scale=1.0*, *size=None*)

Draw samples from the Laplace or double exponential distribution with specified location (or mean) and scale (decay).

The Laplace distribution is similar to the Gaussian/normal distribution, but is sharper at the peak and has fatter tails. It represents the difference between two independent, identically distributed exponential random variables.

**Parameters**

**loc**

[float or array_like of floats, optional] The position, $\mu$, of the distribution peak. Default is 0.

**scale**

[float or array_like of floats, optional] $\lambda$, the exponential decay. Default is 1. Must be non-negative.

**size**

[int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `loc` and `scale` are both scalars. Otherwise, `np.broadcast(loc, scale).size` samples are drawn.

**Returns**

**out**

[ndarray or scalar] Drawn samples from the parameterized Laplace distribution.

## Notes

It has the probability density function

$$f(x;\mu,\lambda) = \frac{1}{2\lambda} \exp\left(-\frac{|x-\mu|}{\lambda}\right).$$

The first law of Laplace, from 1774, states that the frequency of an error can be expressed as an exponential function of the absolute magnitude of the error, which leads to the Laplace distribution. For many problems in economics and health sciences, this distribution seems to model the data better than the standard Gaussian distribution.

### References

[1], [2], [3], [4]

### Examples

Draw samples from the distribution

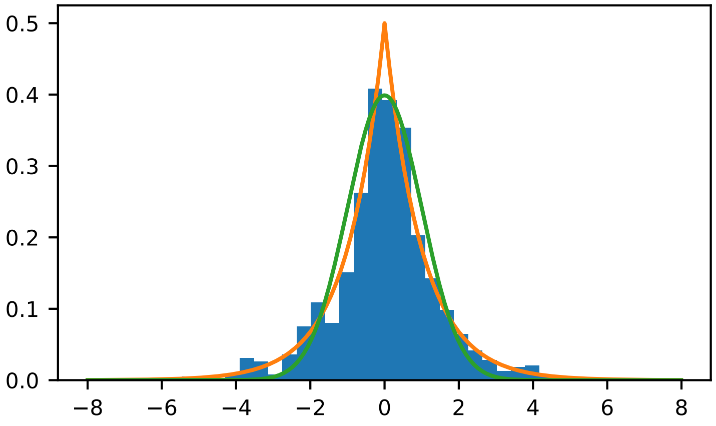
```
>>> loc, scale = 0., 1.
>>> s = np.random.default_rng().laplace(loc, scale, 1000)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 30, density=True)
>>> x = np.arange(-8., 8., .01)
>>> pdf = np.exp(-abs(x-loc)/scale)/(2.*scale)
>>> plt.plot(x, pdf)
```

Plot Gaussian for comparison:

```
>>> g = (1/(scale * np.sqrt(2 * np.pi)) *
...      np.exp(-(x - loc)**2 / (2 * scale**2)))
>>> plt.plot(x,g)
```



method

`Generator.`**`logistic`**(*loc=0.0*, *scale=1.0*, *size=None*)

Draw samples from a logistic distribution.

Samples are drawn from a logistic distribution with specified parameters, loc (location or mean, also median), and scale (>0).

**Parameters**

**loc**

[float or array_like of floats, optional] Parameter of the distribution. Default is 0.

**scale**

[float or array_like of floats, optional] Parameter of the distribution. Must be non-negative. Default is 1.

**size**

[int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `loc` and `scale` are both scalars. Otherwise, `np.broadcast(loc, scale).size` samples are drawn.

**Returns**

**out**

[ndarray or scalar] Drawn samples from the parameterized logistic distribution.

**See also:**

`scipy.stats.logistic`

probability density function, distribution or cumulative density function, etc.

## Notes

The probability density for the Logistic distribution is

$$P(x) = P(x) = \frac{e^{-(x-\mu)/s}}{s(1+e^{-(x-\mu)/s})^2},$$

where $\mu$ = location and $s$ = scale.

The Logistic distribution is used in Extreme Value problems where it can act as a mixture of Gumbel distributions, in Epidemiology, and by the World Chess Federation (FIDE) where it is used in the Elo ranking system, assuming the performance of each player is a logistically distributed random variable.

## References

[1], [2], [3]

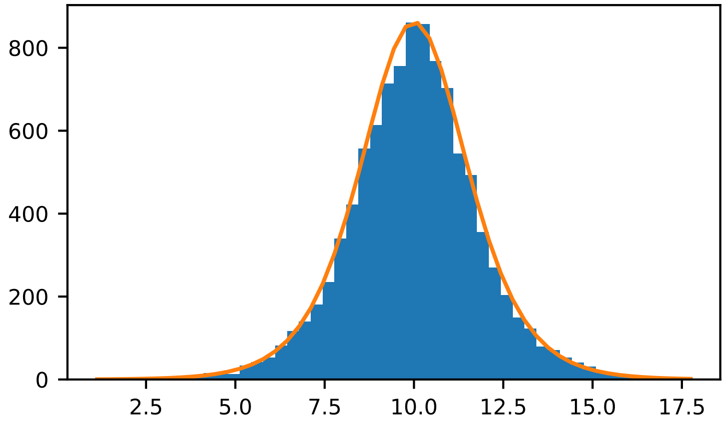
## Examples

Draw samples from the distribution:

```
>>> loc, scale = 10, 1
>>> s = np.random.default_rng().logistic(loc, scale, 10000)
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, bins=50)
```

# plot against distribution

```
>>> def logist(x, loc, scale):
...     return np.exp((loc-x)/scale)/(scale*(1+np.exp((loc-x)/scale))**2)
>>> lgst_val = logist(bins, loc, scale)
>>> plt.plot(bins, lgst_val * count.max() / lgst_val.max())
>>> plt.show()
```



method

`Generator.`**`lognormal`**(*mean=0.0*, *sigma=1.0*, *size=None*)

Draw samples from a log-normal distribution.

Draw samples from a log-normal distribution with specified mean, standard deviation, and array shape. Note that the mean and standard deviation are not the values for the distribution itself, but of the underlying normal distribution it is derived from.

**Parameters**

**mean**

[float or array_like of floats, optional] Mean value of the underlying normal distribution. Default is 0.

**sigma**

[float or array_like of floats, optional] Standard deviation of the underlying normal distribution. Must be non-negative. Default is 1.

**size**

[int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `mean` and `sigma` are both scalars. Otherwise, `np.broadcast(mean, sigma).size` samples are drawn.

**Returns**

**out**

[ndarray or scalar] Drawn samples from the parameterized log-normal distribution.

**See also:**

**`scipy.stats.lognorm`**

probability density function, distribution, cumulative density function, etc.

### Notes

A variable *x* has a log-normal distribution if *log(x)* is normally distributed. The probability density function for the log-normal distribution is:

$$p(x) = \frac{1}{\sigma x \sqrt{2\pi}} e^{(-\frac{(ln(x)-\mu)^2}{2\sigma^2})}$$

where $\mu$ is the mean and $\sigma$ is the standard deviation of the normally distributed logarithm of the variable. A log-normal distribution results if a random variable is the *product* of a large number of independent, identically-distributed variables in the same way that a normal distribution results if the variable is the *sum* of a large number of independent, identically-distributed variables.

### References

[1], [2]

### Examples

Draw samples from the distribution:

```
>>> rng = np.random.default_rng()
>>> mu, sigma = 3., 1. # mean and standard deviation
>>> s = rng.lognormal(mu, sigma, 1000)
```

Display the histogram of the samples, along with the probability density function:

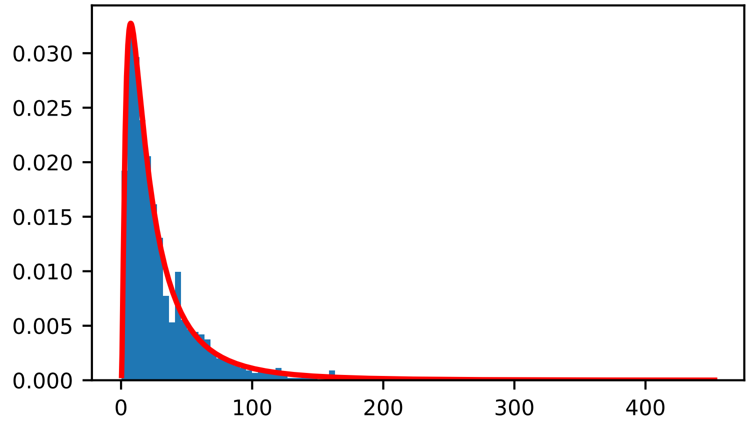
```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 100, density=True, align='mid')
```

```
>>> x = np.linspace(min(bins), max(bins), 10000)
>>> pdf = (np.exp(-(np.log(x) - mu)**2 / (2 * sigma**2))
...        / (x * sigma * np.sqrt(2 * np.pi)))
```

```
>>> plt.plot(x, pdf, linewidth=2, color='r')
>>> plt.axis('tight')
>>> plt.show()
```

Demonstrate that taking the products of random samples from a uniform distribution can be fit well by a log-normal probability density function.

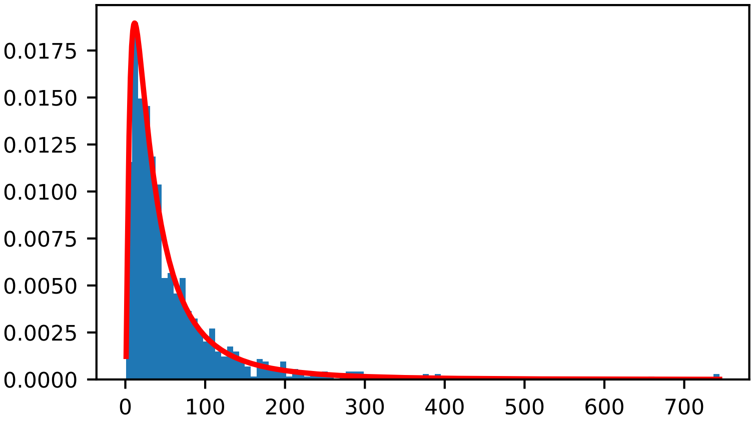
```
>>> # Generate a thousand samples: each is the product of 100 random
>>> # values, drawn from a normal distribution.
>>> rng = rng
>>> b = []
>>> for i in range(1000):
...    a = 10. + rng.standard_normal(100)
...    b.append(np.product(a))
```



```
>>> b = np.array(b) / np.min(b) # scale values to be positive
>>> count, bins, ignored = plt.hist(b, 100, density=True, align='mid')
>>> sigma = np.std(np.log(b))
>>> mu = np.mean(np.log(b))
```

```
>>> x = np.linspace(min(bins), max(bins), 10000)
>>> pdf = (np.exp(-(np.log(x) - mu)**2 / (2 * sigma**2))
...        / (x * sigma * np.sqrt(2 * np.pi)))
```

```
>>> plt.plot(x, pdf, color='r', linewidth=2)
>>> plt.show()
```



method

`Generator.`**`logseries`**(*p*, *size=None*)

Draw samples from a logarithmic series distribution.

Samples are drawn from a log series distribution with specified shape parameter, 0 < `p` < 1.

**Parameters**

**p**

[float or array_like of floats] Shape parameter for the distribution. Must be in the range (0, 1).

**size**

[int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `p` is a scalar. Otherwise, `np.array(p).size` samples are drawn.

**Returns**

**out**

[ndarray or scalar] Drawn samples from the parameterized logarithmic series distribution.

**See also:**

**`scipy.stats.logser`**

probability density function, distribution or cumulative density function, etc.

## Notes

The probability mass function for the Log Series distribution is

$$P(k) = \frac{-p^k}{k \ln(1-p)},$$

where p = probability.

The log series distribution is frequently used to represent species richness and occurrence, first proposed by Fisher, Corbet, and Williams in 1943 [2]. It may also be used to model the numbers of occupants seen in cars [3].

## References

[1], [2], [3], [4]

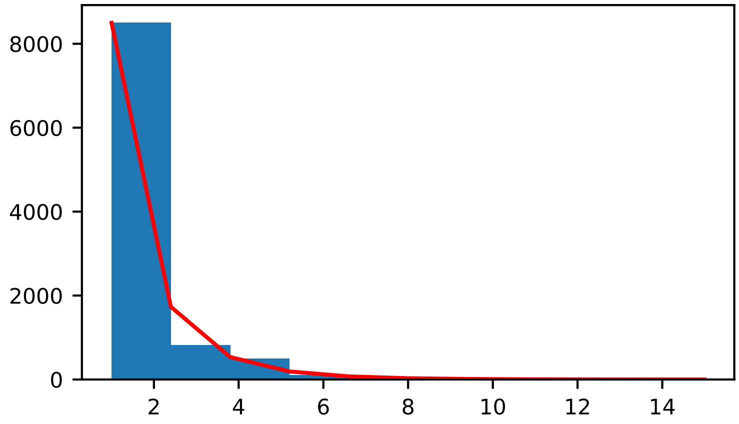
## Examples

Draw samples from the distribution:

```
>>> a = .6
>>> s = np.random.default_rng().logseries(a, 10000)
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s)
```

# plot against distribution

```
>>> def logseries(k, p):
...     return -p**k/(k*np.log(1-p))
>>> plt.plot(bins, logseries(bins, a) * count.max()/
...          logseries(bins, a).max(), 'r')
>>> plt.show()
```



method

**Generator.multinomial**(*n*, *pvals*, *size=None*)

Draw samples from a multinomial distribution.

The multinomial distribution is a multivariate generalization of the binomial distribution. Take an experiment with one of `p` possible outcomes. An example of such an experiment is throwing a dice, where the outcome can be 1 through 6. Each sample drawn from the distribution represents *n* such experiments. Its values, `X_i = [X_0, X_1, ..., X_p]`, represent the number of times the outcome was `i`.

**Parameters**

**n**

[int or array-like of ints] Number of experiments.

**pvals**

[sequence of floats, length p] Probabilities of each of the `p` different outcomes. These must sum to 1 (however, the last element is always assumed to account for the remaining probability, as long as `sum(pvals[:-1]) <= 1)`.

**size**

[int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. Default is None, in which case a single value is returned.

**Returns**

**out**

[ndarray] The drawn samples, of shape *size*, if that was provided. If not, the shape is `(N,)`.

In other words, each entry `out[i,j,...,:]` is an N-dimensional value drawn from the distribution.

### Examples

Throw a dice 20 times:

```
>>> rng = np.random.default_rng()
>>> rng.multinomial(20, [1/6.]*6, size=1)
array([[4, 1, 7, 5, 2, 1]])  # random
```

It landed 4 times on 1, once on 2, etc.

Now, throw the dice 20 times, and 20 times again:

```
>>> rng.multinomial(20, [1/6.]*6, size=2)
array([[3, 4, 3, 3, 4, 3],
       [2, 4, 3, 4, 0, 7]])  # random
```

For the first run, we threw 3 times 1, 4 times 2, etc. For the second, we threw 2 times 1, 4 times 2, etc.

Now, do one experiment throwing the dice 10 time, and 10 times again, and another throwing the dice 20 times, and 20 times again:

```
>>> rng.multinomial([[10], [20]], [1/6.]*6, size=2)
array([[[2, 4, 0, 1, 2, 1],
        [1, 3, 0, 3, 1, 2]],
       [[1, 4, 4, 4, 4, 3],
        [3, 3, 2, 5, 5, 2]]])  # random
```

The first array shows the outcomes of throwing the dice 10 times, and the second shows the outcomes from throwing the dice 20 times.

A loaded die is more likely to land on number 6:

```
>>> rng.multinomial(100, [1/7.]*5 + [2/7.])
array([11, 16, 14, 17, 16, 26])  # random
```

The probability inputs should be normalized. As an implementation detail, the value of the last entry is ignored and assumed to take up any leftover probability mass, but this should not be relied on. A biased coin which has twice as much weight on one side as on the other should be sampled like so:

```
>>> rng.multinomial(100, [1.0 / 3, 2.0 / 3])  # RIGHT
array([38, 62])  # random
```

not like:

```
>>> rng.multinomial(100, [1.0, 2.0])  # WRONG
Traceback (most recent call last):
ValueError: pvals < 0, pvals > 1 or pvals contains NaNs
```

method

`Generator.`**`multivariate_hypergeometric`**(*colors*, *nsample*, *size=None*, *method='marginals'*)

Generate variates from a multivariate hypergeometric distribution.

The multivariate hypergeometric distribution is a generalization of the hypergeometric distribution.

Choose `nsample` items at random without replacement from a collection with `N` distinct types. `N` is the length of `colors`, and the values in `colors` are the number of occurrences of that type in the collection. The total number of items in the collection is `sum(colors)`. Each random variate generated by this function is a vector of length `N` holding the counts of the different types that occurred in the `nsample` items.

The name `colors` comes from a common description of the distribution: it is the probability distribution of the number of marbles of each color selected without replacement from an urn containing marbles of different colors; `colors[i]` is the number of marbles in the urn with color `i`.

**Parameters**

**colors**

[sequence of integers] The number of each type of item in the collection from which a sample is drawn. The values in `colors` must be nonnegative. To avoid loss of precision in the algorithm, `sum(colors)` must be less than `10**9` when *method* is "marginals".

**nsample**

[int] The number of items selected. `nsample` must not be greater than `sum(colors)`.

**size**

[int or tuple of ints, optional] The number of variates to generate, either an integer or a tuple holding the shape of the array of variates. If the given size is, e.g., `(k, m)`, then `k * m` variates are drawn, where one variate is a vector of length `len(colors)`, and the return value has shape `(k, m, len(colors))`. If *size* is an integer, the output has shape `(size, len(colors))`. Default is None, in which case a single variate is returned as an array with shape `(len(colors),)`.

**method**

[string, optional] Specify the algorithm that is used to generate the variates. Must be 'count' or 'marginals' (the default). See the Notes for a description of the methods.

**Returns**

**variates**

[ndarray] Array of variates drawn from the multivariate hypergeometric distribution.

**See also:**

**`hypergeometric`**

Draw samples from the (univariate) hypergeometric distribution.

## Notes

The two methods do not return the same sequence of variates.

The "count" algorithm is roughly equivalent to the following numpy code:

```
choices = np.repeat(np.arange(len(colors)), colors)
selection = np.random.choice(choices, nsample, replace=False)
variate = np.bincount(selection, minlength=len(colors))
```

The "count" algorithm uses a temporary array of integers with length `sum(colors)`.

The "marginals" algorithm generates a variate by using repeated calls to the univariate hypergeometric sampler. It is roughly equivalent to:

```
variate = np.zeros(len(colors), dtype=np.int64)
# `remaining` is the cumulative sum of `colors` from the last
# element to the first; e.g. if `colors` is [3, 1, 5], then
# `remaining` is [9, 6, 5].
remaining = np.cumsum(colors[::-1])[::-1]
for i in range(len(colors)-1):
    if nsample < 1:
        break
    variate[i] = hypergeometric(colors[i], remaining[i+1],
                               nsample)
    nsample -= variate[i]
variate[-1] = nsample
```

The default method is "marginals". For some cases (e.g. when *colors* contains relatively small integers), the "count" method can be significantly faster than the "marginals" method. If performance of the algorithm is important, test the two methods with typical inputs to decide which works best.

New in version 1.18.0.

### Examples

```
>>> colors = [16, 8, 4]
>>> seed = 4861946401452
>>> gen = np.random.Generator(np.random.PCG64(seed))
>>> gen.multivariate_hypergeometric(colors, 6)
array([5, 0, 1])
>>> gen.multivariate_hypergeometric(colors, 6, size=3)
array([[5, 0, 1],
       [2, 2, 2],
       [3, 3, 0]])
>>> gen.multivariate_hypergeometric(colors, 6, size=(2, 2))
array([[[3, 2, 1],
        [3, 2, 1]],
       [[4, 1, 1],
        [3, 2, 1]]])
```

method

`Generator.`**`multivariate_normal`**(*mean*, *cov*, *size=None*, *check_valid='warn'*, *tol=1e-8*)

Draw random samples from a multivariate normal distribution.

The multivariate normal, multinormal or Gaussian distribution is a generalization of the one-dimensional normal distribution to higher dimensions. Such a distribution is specified by its mean and covariance matrix. These parameters are analogous to the mean (average or "center") and variance (standard deviation, or "width," squared) of the one-dimensional normal distribution.

**Parameters**

**mean**

[1-D array_like, of length N] Mean of the N-dimensional distribution.

**cov**

[2-D array_like, of shape (N, N)] Covariance matrix of the distribution. It must be symmetric and positive-semidefinite for proper sampling.

**size**

[int or tuple of ints, optional] Given a shape of, for example, `(m,n,k)`, `m*n*k` samples are generated, and packed in an *m*-by-*n*-by-*k* arrangement. Because each sample is *N*-dimensional, the output shape is `(m,n,k,N)`. If no shape is specified, a single (*N*-D) sample is returned.

**check_valid**

[{ 'warn', 'raise', 'ignore' }, optional] Behavior when the covariance matrix is not positive semidefinite.

**tol**

[float, optional] Tolerance when checking the singular values in covariance matrix. cov is cast to double before the check.

**method**

[{ 'svd', 'eigh', 'cholesky'}, optional] The cov input is used to compute a factor matrix A such that `A @ A.T = cov`. This argument is used to select the method used to compute the factor matrix A. The default method 'svd' is the slowest, while 'cholesky' is the fastest but less robust than the slowest method. The method *eigh* uses eigen decomposition to compute A and is faster than svd but slower than cholesky.

New in version 1.18.0.

**Returns**

**out**

[ndarray] The drawn samples, of shape *size*, if that was provided. If not, the shape is `(N,)`.

In other words, each entry `out[i,j,...,:]` is an N-dimensional value drawn from the distribution.

## Notes

The mean is a coordinate in N-dimensional space, which represents the location where samples are most likely to be generated. This is analogous to the peak of the bell curve for the one-dimensional or univariate normal distribution.

Covariance indicates the level to which two variables vary together. From the multivariate normal distribution, we draw N-dimensional samples, $X = [x_1, x_2, ... x_N]$. The covariance matrix element $C_{ij}$ is the covariance of $x_i$ and $x_j$. The element $C_{ii}$ is the variance of $x_i$ (i.e. its "spread").

Instead of specifying the full covariance matrix, popular approximations include:

- Spherical covariance (*cov* is a multiple of the identity matrix)
- Diagonal covariance (*cov* has non-negative elements, and only on the diagonal)

This geometrical property can be seen in two dimensions by plotting generated data-points:

```
>>> mean = [0, 0]
>>> cov = [[1, 0], [0, 100]]  # diagonal covariance
```

Diagonal covariance means that points are oriented along x or y-axis:

```
>>> import matplotlib.pyplot as plt
>>> x, y = np.random.default_rng().multivariate_normal(mean, cov, 5000).T
>>> plt.plot(x, y, 'x')
>>> plt.axis('equal')
>>> plt.show()
```

Note that the covariance matrix must be positive semidefinite (a.k.a. nonnegative-definite). Otherwise, the behavior of this method is undefined and backwards compatibility is not guaranteed.

### References

[1], [2]

### Examples

```
>>> mean = (1, 2)
>>> cov = [[1, 0], [0, 1]]
>>> rng = np.random.default_rng()
>>> x = rng.multivariate_normal(mean, cov, (3, 3))
>>> x.shape
(3, 3, 2)
```

We can use a different method other than the default to factorize cov: »> y = rng.multivariate_normal(mean, cov, (3, 3), method='cholesky') »> y.shape (3, 3, 2)

The following is probably true, given that 0.6 is roughly twice the standard deviation:

```
>>> list((x[0,0,:] - mean) < 0.6)
[True, True] # random
```

method

`Generator.`**`negative_binomial`**(*n*, *p*, *size=None*)

Draw samples from a negative binomial distribution.

Samples are drawn from a negative binomial distribution with specified parameters, *n* successes and *p* probability of success where *n* is > 0 and *p* is in the interval (0, 1].

**Parameters**

**n**

[float or array_like of floats] Parameter of the distribution, > 0.

**p**

[float or array_like of floats] Parameter of the distribution. Must satisfy 0 < p <= 1.

**size**

[int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `n` and `p` are both scalars. Otherwise, `np.broadcast(n, p).size` samples are drawn.

**Returns**

**out**

[ndarray or scalar] Drawn samples from the parameterized negative binomial distribution, where each sample is equal to N, the number of failures that occurred before a total of n successes was reached.

### Notes

The probability mass function of the negative binomial distribution is

$$P(N;n,p) = \frac{\Gamma(N+n)}{N!\Gamma(n)}p^{n}(1-p)^{N},$$

where $n$ is the number of successes, $p$ is the probability of success, $N+n$ is the number of trials, and $\Gamma$ is the gamma function. When $n$ is an integer, $\frac{\Gamma(N+n)}{N!\Gamma(n)} = \binom{N+n-1}{N}$, which is the more common form of this term in the the pmf. The negative binomial distribution gives the probability of N failures given n successes, with a success on the last trial.

If one throws a die repeatedly until the third time a "1" appears, then the probability distribution of the number of non-"1"s that appear before the third "1" is a negative binomial distribution.

### References

[1], [2]

### Examples

Draw samples from the distribution:

A real world example. A company drills wild-cat oil exploration wells, each with an estimated probability of success of 0.1. What is the probability of having one success for each successive well, that is what is the probability of a single success after drilling 5 wells, after 6 wells, etc.?

```
>>> s = np.random.default_rng().negative_binomial(1, 0.1, 100000)
>>> for i in range(1, 11):
...     probability = sum(s<i) / 100000.
...     print(i, "wells drilled, probability of one success =", probability)
```

method

`Generator.`**`noncentral_chisquare`**(*df*, *nonc*, *size=None*)

Draw samples from a noncentral chi-square distribution.

The noncentral $\chi^2$ distribution is a generalization of the $\chi^2$ distribution.

**Parameters**

**df**

[float or array_like of floats] Degrees of freedom, must be > 0.

Changed in version 1.10.0: Earlier NumPy versions required dfnum > 1.

**nonc**

[float or array_like of floats] Non-centrality, must be non-negative.

**size**

[int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `df` and `nonc` are both scalars. Otherwise, `np.broadcast(df, nonc).size` samples are drawn.

**Returns**

**out**

[ndarray or scalar] Drawn samples from the parameterized noncentral chi-square distribution.

## Notes

The probability density function for the noncentral Chi-square distribution is

$$P(x;df,nonc) = \sum_{i=0}^{\infty} \frac{e^{-nonc/2}(nonc/2)^{i}}{i!} P_{Y_{df+2i}}(x),$$

where $Y_q$ is the Chi-square with q degrees of freedom.

## References

[1]

## Examples

Draw values from the distribution and plot the histogram

```
>>> rng = np.random.default_rng()
>>> import matplotlib.pyplot as plt
>>> values = plt.hist(rng.noncentral_chisquare(3, 20, 100000),
...                   bins=200, density=True)
>>> plt.show()
```

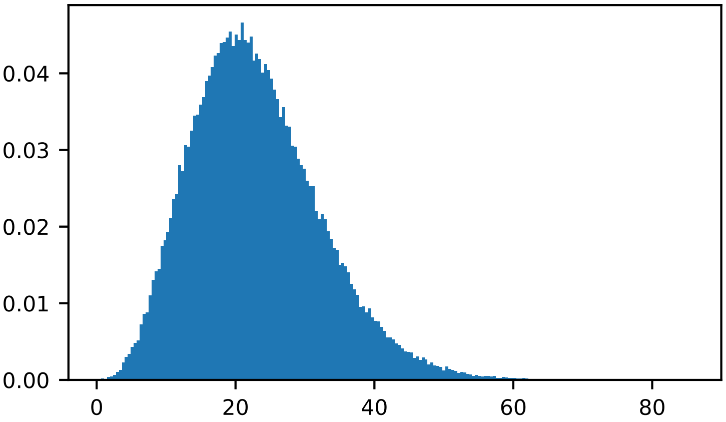
Draw values from a noncentral chisquare with very small noncentrality, and compare to a chisquare.

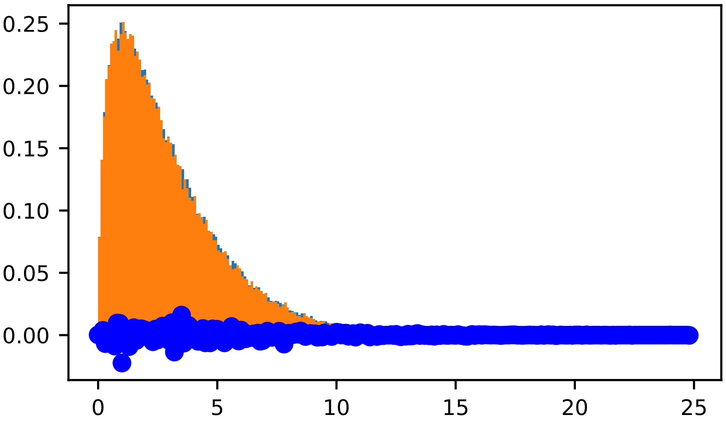
```
>>> plt.figure()
>>> values = plt.hist(rng.noncentral_chisquare(3, .0000001, 100000),
...                   bins=np.arange(0., 25, .1), density=True)
>>> values2 = plt.hist(rng.chisquare(3, 100000),
...                    bins=np.arange(0., 25, .1), density=True)
>>> plt.plot(values[1][0:-1], values[0]-values2[0], 'ob')
>>> plt.show()
```

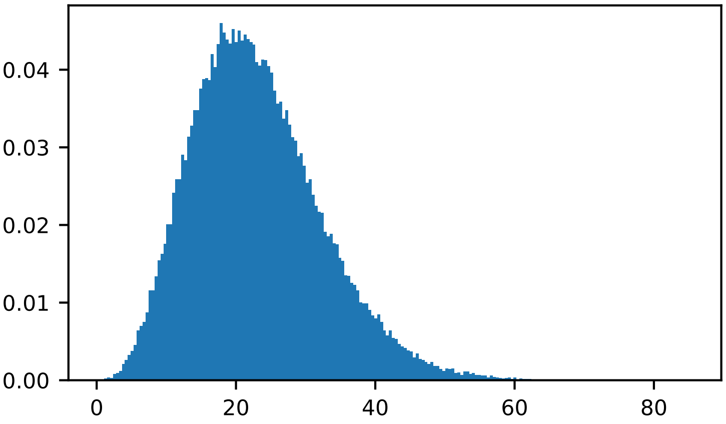
Demonstrate how large values of non-centrality lead to a more symmetric distribution.

```
>>> plt.figure()
>>> values = plt.hist(rng.noncentral_chisquare(3, 20, 100000),
...                   bins=200, density=True)
>>> plt.show()
```

method


0.04
0.03
0.02
0.01
0.00
0
20
40
60
80


0.25
0.20
0.15
0.10
0.05
0.00
0
5
10
15
20
25



`Generator.`**`noncentral_f`**(*dfnum*, *dfden*, *nonc*, *size=None*)

Draw samples from the noncentral F distribution.

Samples are drawn from an F distribution with specified parameters, *dfnum* (degrees of freedom in numerator) and *dfden* (degrees of freedom in denominator), where both parameters > 1. *nonc* is the non-centrality parameter.

**Parameters**

**dfnum**

[float or array_like of floats] Numerator degrees of freedom, must be > 0.

Changed in version 1.14.0: Earlier NumPy versions required dfnum > 1.

**dfden**

[float or array_like of floats] Denominator degrees of freedom, must be > 0.

**nonc**

[float or array_like of floats] Non-centrality parameter, the sum of the squares of the numerator means, must be >= 0.

**size**

[int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `dfnum`, `dfden`, and `nonc` are all scalars. Otherwise, `np.broadcast(dfnum, dfden, nonc).size` samples are drawn.

**Returns**

**out**

[ndarray or scalar] Drawn samples from the parameterized noncentral Fisher distribution.

### Notes

When calculating the power of an experiment (power = probability of rejecting the null hypothesis when a specific alternative is true) the non-central F statistic becomes important. When the null hypothesis is true, the F statistic follows a central F distribution. When the null hypothesis is not true, then it follows a non-central F statistic.

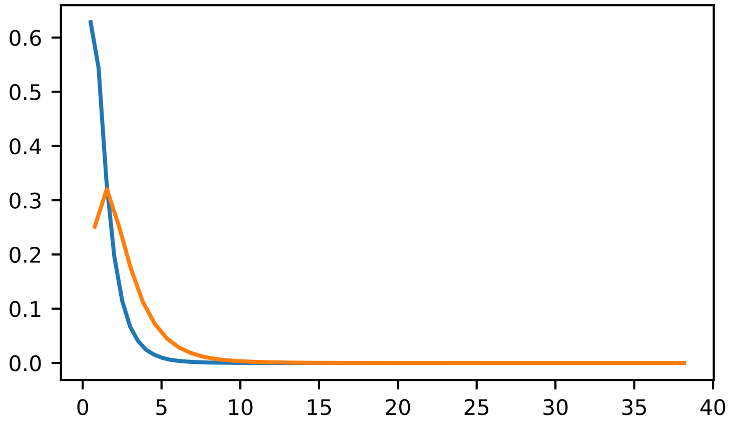
### References

[1], [2]

### Examples

In a study, testing for a specific alternative to the null hypothesis requires use of the Noncentral F distribution. We need to calculate the area in the tail of the distribution that exceeds the value of the F distribution for the null hypothesis. We'll plot the two probability distributions for comparison.

```
>>> rng = np.random.default_rng()
>>> dfnum = 3 # between group deg of freedom
>>> dfden = 20 # within groups degrees of freedom
>>> nonc = 3.0
>>> nc_vals = rng.noncentral_f(dfnum, dfden, nonc, 1000000)
>>> NF = np.histogram(nc_vals, bins=50, density=True)
>>> c_vals = rng.f(dfnum, dfden, 1000000)
>>> F = np.histogram(c_vals, bins=50, density=True)
>>> import matplotlib.pyplot as plt
>>> plt.plot(F[1][1:], F[0])
>>> plt.plot(NF[1][1:], NF[0])
>>> plt.show()
```



method

`Generator.normal`(*loc=0.0*, *scale=1.0*, *size=None*)
Draw random samples from a normal (Gaussian) distribution.

The probability density function of the normal distribution, first derived by De Moivre and 200 years later by both Gauss and Laplace independently [2], is often called the bell curve because of its characteristic shape (see the example below).

The normal distributions occurs often in nature. For example, it describes the commonly occurring distribution of samples influenced by a large number of tiny, random disturbances, each with its own unique distribution [2].

**Parameters**

**loc**

[float or array_like of floats] Mean ("centre") of the distribution.

**scale**

[float or array_like of floats] Standard deviation (spread or "width") of the distribution. Must be non-negative.

**size**

[int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `loc` and `scale` are both scalars. Otherwise, `np.broadcast(loc, scale).size` samples are drawn.

**Returns**

**out**

[ndarray or scalar] Drawn samples from the parameterized normal distribution.

**See also:**

**`scipy.stats.norm`**

probability density function, distribution or cumulative density function, etc.

## Notes

The probability density for the Gaussian distribution is

$$p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}},$$

where $\mu$ is the mean and $\sigma$ the standard deviation. The square of the standard deviation, $\sigma^2$, is called the variance.

The function has its peak at the mean, and its "spread" increases with the standard deviation (the function reaches 0.607 times its maximum at $x+\sigma$ and $x-\sigma$ [2]). This implies that `normal` is more likely to return samples lying close to the mean, rather than those far away.

### References

[1], [2]

### Examples

Draw samples from the distribution:

```
>>> mu, sigma = 0, 0.1 # mean and standard deviation
>>> s = np.random.default_rng().normal(mu, sigma, 1000)
```

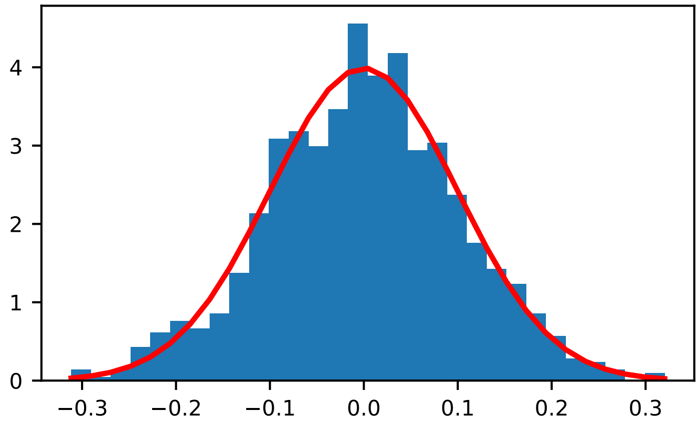
Verify the mean and the variance:

```
>>> abs(mu - np.mean(s))
0.0  # may vary
```

```
>>> abs(sigma - np.std(s, ddof=1))
0.1  # may vary
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 30, density=True)
>>> plt.plot(bins, 1/(sigma * np.sqrt(2 * np.pi)) *
...                np.exp( - (bins - mu)**2 / (2 * sigma**2) ),
...          linewidth=2, color='r')
>>> plt.show()
```



Two-by-four array of samples from N(3, 6.25):

```
>>> np.random.default_rng().normal(3, 2.5, size=(2, 4))
array([[-4.49401501,  4.00950034, -1.81814867,  7.29718677],   # random
       [ 0.39924804,  4.68456316,  4.99394529,  4.84057254]])  # random
```

method

`Generator.`**`pareto`**(*a*, *size=None*)

Draw samples from a Pareto II or Lomax distribution with specified shape.

The Lomax or Pareto II distribution is a shifted Pareto distribution. The classical Pareto distribution can be obtained from the Lomax distribution by adding 1 and multiplying by the scale parameter `m` (see Notes). The smallest value of the Lomax distribution is zero while for the classical Pareto distribution it is `mu`, where the standard Pareto distribution has location `mu = 1`. Lomax can also be considered as a simplified version of the Generalized Pareto distribution (available in SciPy), with the scale set to one and the location set to zero.

The Pareto distribution must be greater than zero, and is unbounded above. It is also known as the "80-20 rule". In this distribution, 80 percent of the weights are in the lowest 20 percent of the range, while the other 20 percent fill the remaining 80 percent of the range.

**Parameters**

**a**

[float or array_like of floats] Shape of the distribution. Must be positive.

**size**

[int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `a` is a scalar. Otherwise, `np.array(a).size` samples are drawn.

**Returns**

**out**

[ndarray or scalar] Drawn samples from the parameterized Pareto distribution.

**See also:**

**`scipy.stats.lomax`**

probability density function, distribution or cumulative density function, etc.

**`scipy.stats.genpareto`**

probability density function, distribution or cumulative density function, etc.

## Notes

The probability density for the Pareto distribution is

$$p(x) = \frac{am^a}{x^{a+1}}$$

where $a$ is the shape and $m$ the scale.

The Pareto distribution, named after the Italian economist Vilfredo Pareto, is a power law probability distribution useful in many real world problems. Outside the field of economics it is generally referred to as the Bradford distribution. Pareto developed the distribution to describe the distribution of wealth in an economy. It has also found use in insurance, web page access statistics, oil field sizes, and many other problems, including the download frequency for projects in Sourceforge [1]. It is one of the so-called "fat-tailed" distributions.

### References

[1], [2], [3], [4]

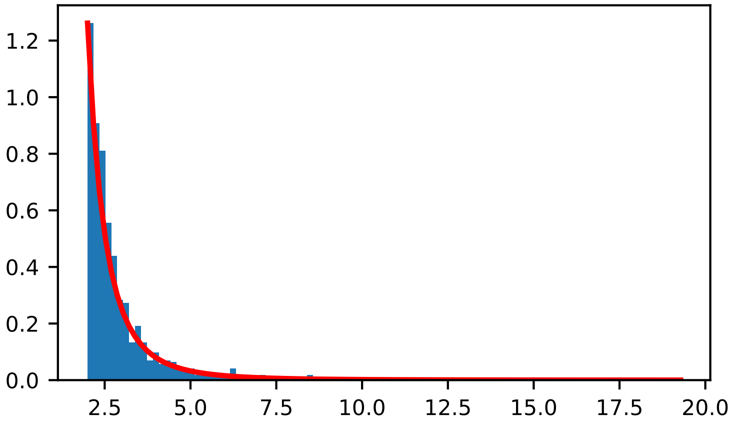
### Examples

Draw samples from the distribution:

```
>>> a, m = 3., 2.  # shape and mode
>>> s = (np.random.default_rng().pareto(a, 1000) + 1) * m
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, _ = plt.hist(s, 100, density=True)
>>> fit = a*m**a / bins**(a+1)
>>> plt.plot(bins, max(count)*fit/max(fit), linewidth=2, color='r')
>>> plt.show()
```



method

`Generator.`**`poisson`**(*lam=1.0*, *size=None*)

Draw samples from a Poisson distribution.

The Poisson distribution is the limit of the binomial distribution for large N.

**Parameters**

**lam**

[float or array_like of floats] Expectation of interval, must be >= 0. A sequence of expectation intervals must be broadcastable over the requested size.

**size**

[int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `lam` is a scalar. Otherwise, `np.array(lam).size` samples are drawn.

**Returns**

**out**

[ndarray or scalar] Drawn samples from the parameterized Poisson distribution.

### Notes

The Poisson distribution

$$f(k;\lambda)=\frac{\lambda^k e^{-\lambda}}{k!}$$

For events with an expected separation $\lambda$ the Poisson distribution $f(k;\lambda)$ describes the probability of $k$ events occurring within the observed interval $\lambda$.

Because the output is limited to the range of the C int64 type, a ValueError is raised when *lam* is within 10 sigma of the maximum representable value.

### References

[1], [2]

### Examples

Draw samples from the distribution:

```
>>> import numpy as np
>>> rng = np.random.default_rng()
>>> s = rng.poisson(5, 10000)
```

Display histogram of the sample:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 14, density=True)
>>> plt.show()
```

Draw each 100 values for lambda 100 and 500:

```
>>> s = rng.poisson(lam=(100., 500.), size=(100, 2))
```

method

`Generator`.**`power`**(*a*, *size=None*)

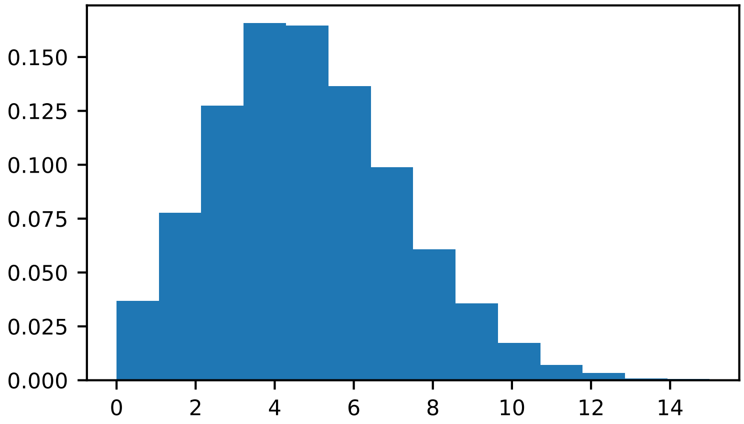
Draws samples in [0, 1] from a power distribution with positive exponent a - 1.

Also known as the power function distribution.

**Parameters**

**a**

[float or array_like of floats] Parameter of the distribution. Must be non-negative.



**size**

[int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `a` is a scalar. Otherwise, `np.array(a).size` samples are drawn.

**Returns**

**out**

[ndarray or scalar] Drawn samples from the parameterized power distribution.

**Raises**

**ValueError**

If a < 1.

## Notes

The probability density function is

$$P(x; a) = ax^{a-1}, 0 \le x \le 1, a > 0.$$

The power function distribution is just the inverse of the Pareto distribution. It may also be seen as a special case of the Beta distribution.

It is used, for example, in modeling the over-reporting of insurance claims.

### References

[1], [2]

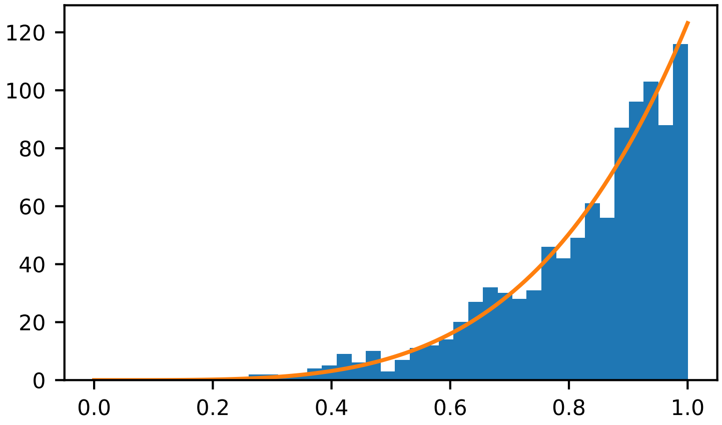
### Examples

Draw samples from the distribution:

```
>>> rng = np.random.default_rng()
>>> a = 5. # shape
>>> samples = 1000
>>> s = rng.power(a, samples)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, bins=30)
>>> x = np.linspace(0, 1, 100)
>>> y = a*x**(a-1.)
>>> normed_y = samples*np.diff(bins)[0]*y
>>> plt.plot(x, normed_y)
>>> plt.show()
```

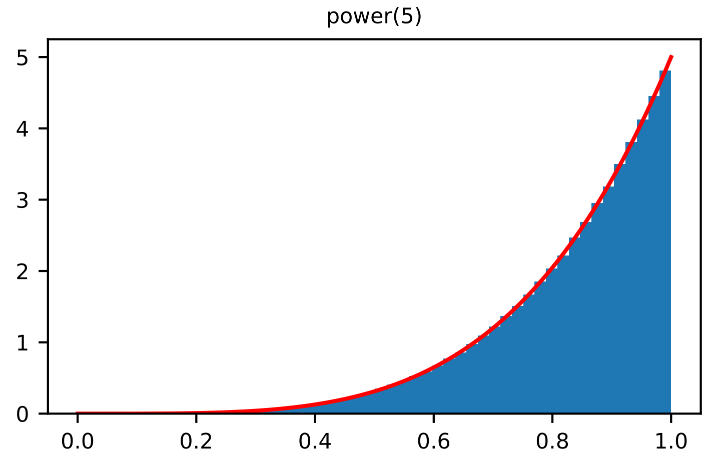


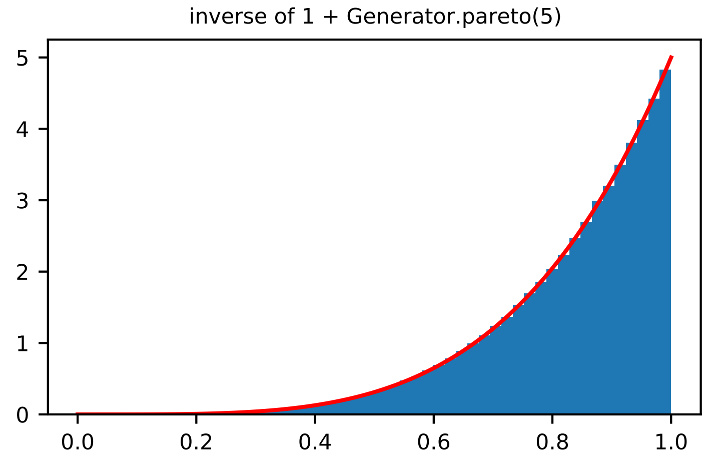
Compare the power function distribution to the inverse of the Pareto.

```
>>> from scipy import stats
>>> rvs = rng.power(5, 1000000)
>>> rvsp = rng.pareto(5, 1000000)
>>> xx = np.linspace(0,1,100)
>>> powpdf = stats.powerlaw.pdf(xx,5)
```

```
>>> plt.figure()
>>> plt.hist(rvs, bins=50, density=True)
>>> plt.plot(xx,powpdf,'r-')
>>> plt.title('power(5)')
>>> plt.figure()
>>> plt.hist(1./(1.+rvsp), bins=50, density=True)
>>> plt.plot(xx,powpdf,'r-')
>>> plt.title('inverse of 1 + Generator.pareto(5)')
```

```
>>> plt.figure()
>>> plt.hist(1./(1.+rvsp), bins=50, density=True)
>>> plt.plot(xx,powpdf,'r-')
>>> plt.title('inverse of stats.pareto(5)')
```



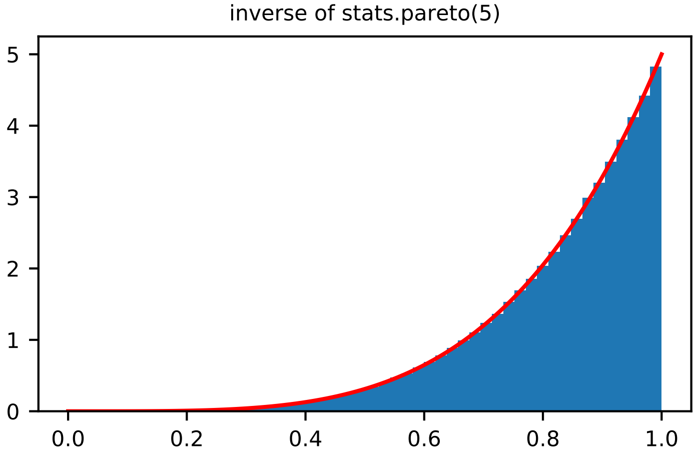


method

`Generator.`**`rayleigh`**(*scale=1.0*, *size=None*)

Draw samples from a Rayleigh distribution.

The $\chi$ and Weibull distributions are generalizations of the Rayleigh.



**Parameters**

**scale**

[float or array_like of floats, optional] Scale, also equals the mode. Must be non-negative. Default is 1.

**size**

[int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `scale` is a scalar. Otherwise, `np.array(scale).size` samples are drawn.

**Returns**

**out**

[ndarray or scalar] Drawn samples from the parameterized Rayleigh distribution.

## Notes

The probability density function for the Rayleigh distribution is

$$P(x;scale) = \frac{x}{scale^2}e^{\frac{-x^2}{2 \cdot scale^2}}$$

The Rayleigh distribution would arise, for example, if the East and North components of the wind velocity had identical zero-mean Gaussian distributions. Then the wind speed would have a Rayleigh distribution.

### References

[1], [2]

### Examples

Draw values from the distribution and plot the histogram

```
>>> from matplotlib.pyplot import hist
>>> rng = np.random.default_rng()
>>> values = hist(rng.rayleigh(3, 100000), bins=200, density=True)
```

Wave heights tend to follow a Rayleigh distribution. If the mean wave height is 1 meter, what fraction of waves are likely to be larger than 3 meters?

```
>>> meanvalue = 1
>>> modevalue = np.sqrt(2 / np.pi) * meanvalue
>>> s = rng.rayleigh(modevalue, 1000000)
```

The percentage of waves larger than 3 meters is:

```
>>> 100.*sum(s>3)/1000000.
0.087300000000000003 # random
```

method

`Generator.`**`standard_cauchy`**(*size=None*)

Draw samples from a standard Cauchy distribution with mode = 0.

Also known as the Lorentz distribution.

**Parameters**

**size**

[int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. Default is None, in which case a single value is returned.

**Returns**

**samples**

[ndarray or scalar] The drawn samples.

### Notes

The probability density function for the full Cauchy distribution is

$$P(x; x_0, \gamma) = \frac{1}{\pi \gamma \bigl[1 + (\frac{x-x_0}{\gamma})^2 \bigr]}$$

and the Standard Cauchy distribution just sets $x_0 = 0$ and $\gamma = 1$

The Cauchy distribution arises in the solution to the driven harmonic oscillator problem, and also describes spectral line broadening. It also describes the distribution of values at which a line tilted at a random angle will cut the x axis.

When studying hypothesis tests that assume normality, seeing how the tests perform on data from a Cauchy distribution is a good indicator of their sensitivity to a heavy-tailed distribution, since the Cauchy looks very much like a Gaussian distribution, but with heavier tails.

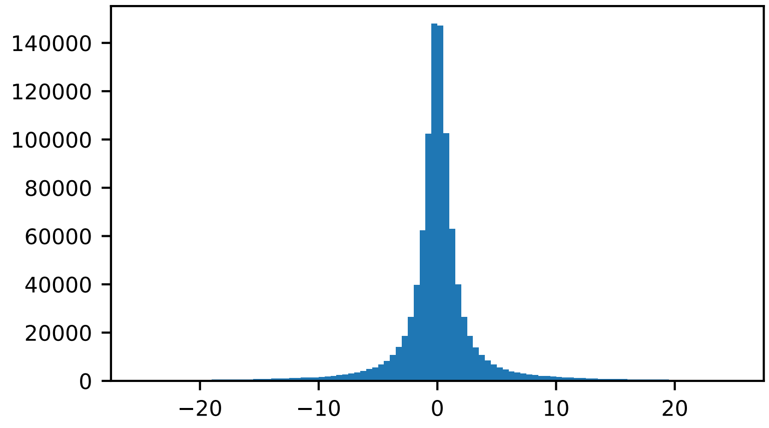
### References

[1], [2], [3]

### Examples

Draw samples and plot the distribution:

```
>>> import matplotlib.pyplot as plt
>>> s = np.random.default_rng().standard_cauchy(1000000)
>>> s = s[(s>-25) & (s<25)]  # truncate distribution so it plots well
>>> plt.hist(s, bins=100)
>>> plt.show()
```



method

`Generator.`**`standard_exponential`**(*size=None*, *dtype=np.float64*, *method='zig'*, *out=None*)

Draw samples from the standard exponential distribution.

*`standard_exponential`* is identical to the exponential distribution with a scale parameter of 1.

**Parameters**

**size**

[int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. Default is None, in which case a single value is returned.

**dtype**

[dtype, optional] Desired dtype of the result, only *float64* and *float32* are supported. Byteorder must be native. The default value is np.float64.

**method**

[str, optional] Either 'inv' or 'zig'. 'inv' uses the default inverse CDF method. 'zig' uses the much faster Ziggurat method of Marsaglia and Tsang.

**out**

[ndarray, optional] Alternative output array in which to place the result. If size is not None, it must have the same shape as the provided size and must match the type of the output values.

**Returns**

**out**

[float or ndarray] Drawn samples.

## Examples

Output a 3x8000 array:

```
>>> n = np.random.default_rng().standard_exponential((3, 8000))
```

method

`Generator.`**`standard_gamma`**(*shape*, *size=None*, *dtype=np.float64*, *out=None*)

Draw samples from a standard Gamma distribution.

Samples are drawn from a Gamma distribution with specified parameters, shape (sometimes designated "k") and scale=1.

**Parameters**

**shape**

[float or array_like of floats] Parameter, must be non-negative.

**size**

[int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `shape` is a scalar. Otherwise, `np.array(shape).size` samples are drawn.

**dtype**

[dtype, optional] Desired dtype of the result, only *float64* and *float32* are supported. Byteorder must be native. The default value is np.float64.

**out**

[ndarray, optional] Alternative output array in which to place the result. If size is not None, it must have the same shape as the provided size and must match the type of the output values.

**Returns**

**out**

[ndarray or scalar] Drawn samples from the parameterized standard gamma distribution.

**See also:**

`scipy.stats.gamma`
probability density function, distribution or cumulative density function, etc.

### Notes

The probability density for the Gamma distribution is

$$p(x) = x^{k-1}\frac{e^{-x/\theta}}{\theta^k\Gamma(k)},$$

where $k$ is the shape and $\theta$ the scale, and $\Gamma$ is the Gamma function.

The Gamma distribution is often used to model the times to failure of electronic components, and arises naturally in processes for which the waiting times between Poisson distributed events are relevant.

### References

[1], [2]

### Examples

Draw samples from the distribution:

```
>>> shape, scale = 2., 1. # mean and width
>>> s = np.random.default_rng().standard_gamma(shape, 1000000)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> import scipy.special as sps
>>> count, bins, ignored = plt.hist(s, 50, density=True)
>>> y = bins**(shape-1) * ((np.exp(-bins/scale))/
...                       (sps.gamma(shape) * scale**shape))
>>> plt.plot(bins, y, linewidth=2, color='r')
>>> plt.show()
```

method

`Generator.`**`standard_normal`**(*size=None*, *dtype=np.float64*, *out=None*)
Draw samples from a standard Normal distribution (mean=0, stdev=1).

**Parameters**

**size**

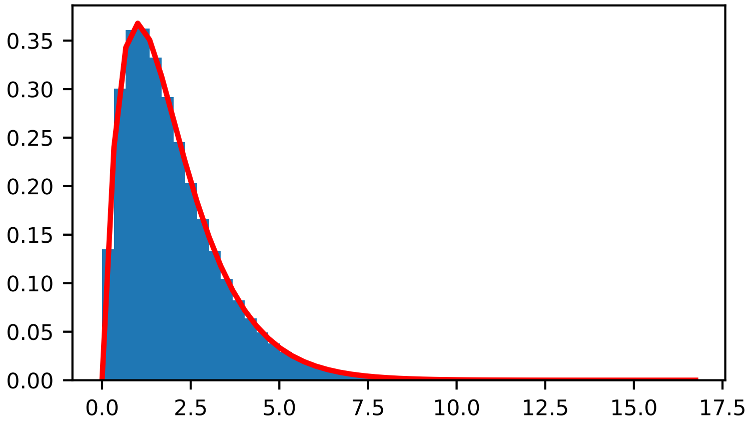
[int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. Default is None, in which case a single value is returned.

**dtype**

[dtype, optional] Desired dtype of the result, only *float64* and *float32* are supported. Byteorder must be native. The default value is np.float64.

**out**

[ndarray, optional] Alternative output array in which to place the result. If size is not None, it must have the same shape as the provided size and must match the type of the output values.



**Returns**

**out**

[float or ndarray] A floating-point array of shape `size` of drawn samples, or a single sample if `size` was not specified.

**See also:**

***normal***

Equivalent function with additional `loc` and `scale` arguments for setting the mean and standard deviation.

## Notes

For random samples from $N(\mu, \sigma^2)$, use one of:

```
mu + sigma * gen.standard_normal(size=...)
gen.normal(mu, sigma, size=...)
```

## Examples

```
>>> rng = np.random.default_rng()
>>> rng.standard_normal()
2.1923875335537315 #random
```

```
>>> s = rng.standard_normal(8000)
>>> s
array([ 0.6888893 ,  0.78096262, -0.89086505, ...,  0.49876311,  # random
       -0.38672696, -0.4685006 ])                                # random
>>> s.shape
(8000,)
>>> s = rng.standard_normal(size=(3, 4, 2))
```

(continues on next page)

(continued from previous page)

```
>>> s.shape
(3, 4, 2)
```

Two-by-four array of samples from $N(3, 6.25)$:

```
>>> 3 + 2.5 * rng.standard_normal(size=(2, 4))
array([[-4.49401501,  4.00950034, -1.81814867,  7.29718677],   # random
       [ 0.39924804,  4.68456316,  4.99394529,  4.84057254]])  # random
```

method

`Generator.`**`standard_t`**(*df*, *size=None*)

Draw samples from a standard Student's t distribution with *df* degrees of freedom.

A special case of the hyperbolic distribution. As *df* gets large, the result resembles that of the standard normal distribution (`standard_normal`).

**Parameters**

**df**

[float or array_like of floats] Degrees of freedom, must be > 0.

**size**

[int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `df` is a scalar. Otherwise, `np.array(df).size` samples are drawn.

**Returns**

**out**

[ndarray or scalar] Drawn samples from the parameterized standard Student's t distribution.

## Notes

The probability density function for the t distribution is

$$P(x, df) = \frac{\Gamma(\frac{df+1}{2})}{\sqrt{\pi df}\Gamma(\frac{df}{2})}\Bigl( 1+\frac{x^2}{df} \Bigr)^{-(df+1)/2}$$

The t test is based on an assumption that the data come from a Normal distribution. The t test provides a way to test whether the sample mean (that is the mean calculated from the data) is a good estimate of the true mean.

The derivation of the t-distribution was first published in 1908 by William Gosset while working for the Guinness Brewery in Dublin. Due to proprietary issues, he had to publish under a pseudonym, and so he used the name Student.

### References

[1], [2]

### Examples

From Dalgaard page 83 [1], suppose the daily energy intake for 11 women in kilojoules (kJ) is:

```
>>> intake = np.array([5260., 5470, 5640, 6180, 6390, 6515, 6805, 7515, \
...                    7515, 8230, 8770])
```

Does their energy intake deviate systematically from the recommended value of 7725 kJ?

We have 10 degrees of freedom, so is the sample mean within 95% of the recommended value?

```
>>> s = np.random.default_rng().standard_t(10, size=100000)
>>> np.mean(intake)
6753.636363636364
>>> intake.std(ddof=1)
1142.1232221373727
```

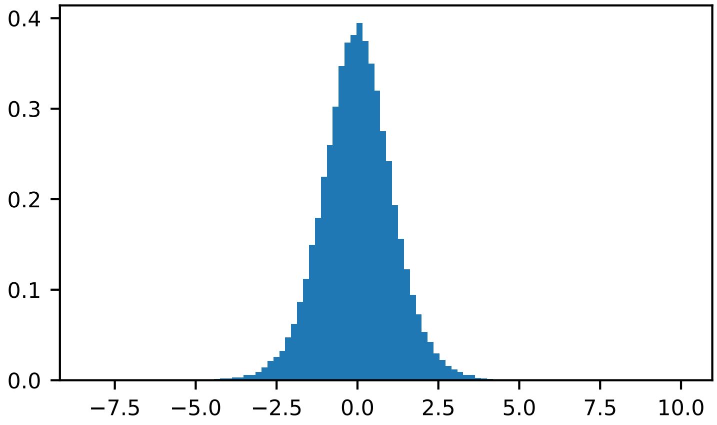
Calculate the t statistic, setting the ddof parameter to the unbiased value so the divisor in the standard deviation will be degrees of freedom, N-1.

```
>>> t = (np.mean(intake)-7725)/(intake.std(ddof=1)/np.sqrt(len(intake)))
>>> import matplotlib.pyplot as plt
>>> h = plt.hist(s, bins=100, density=True)
```

For a one-sided t-test, how far out in the distribution does the t statistic appear?

```
>>> np.sum(s<t) / float(len(s))
0.0090699999999999999  #random
```

So the p-value is about 0.009, which says the null hypothesis has a probability of about 99% of being true.



method

`Generator.`**`triangular`**(*left*, *mode*, *right*, *size=None*)

Draw samples from the triangular distribution over the interval `[left, right]`.

The triangular distribution is a continuous probability distribution with lower limit left, peak at mode, and upper limit right. Unlike the other distributions, these parameters directly define the shape of the pdf.

**Parameters**

**left**

[float or array_like of floats] Lower limit.

**mode**

[float or array_like of floats] The value where the peak of the distribution occurs. The value must fulfill the condition `left <= mode <= right`.

**right**

[float or array_like of floats] Upper limit, must be larger than *left*.

**size**

[int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `left`, `mode`, and `right` are all scalars. Otherwise, `np.broadcast(left, mode, right).size` samples are drawn.

**Returns**

**out**

[ndarray or scalar] Drawn samples from the parameterized triangular distribution.

## Notes

The probability density function for the triangular distribution is

$$P(x;l,m,r) = \begin{cases} \frac{2(x-l)}{(r-l)(m-l)} & \text{for } l \leq x \leq m, \\ \frac{2(r-x)}{(r-l)(r-m)} & \text{for } m \leq x \leq r, \\ 0 & \text{otherwise.} \end{cases}$$

The triangular distribution is often used in ill-defined problems where the underlying distribution is not known, but some knowledge of the limits and mode exists. Often it is used in simulations.

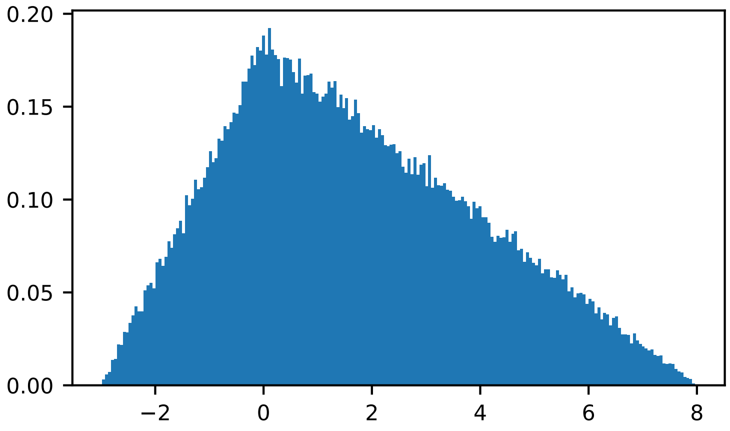
## References

[1]

## Examples

Draw values from the distribution and plot the histogram:

```
>>> import matplotlib.pyplot as plt
>>> h = plt.hist(np.random.default_rng().triangular(-3, 0, 8, 100000), bins=200,
...              density=True)
>>> plt.show()
```



method

`Generator.`**`uniform`**(*low=0.0*, *high=1.0*, *size=None*)

Draw samples from a uniform distribution.

Samples are uniformly distributed over the half-open interval `[low, high)` (includes low, but excludes high). In other words, any value within the given interval is equally likely to be drawn by *`uniform`*.

**Parameters**

**low**

[float or array_like of floats, optional] Lower boundary of the output interval. All values generated will be greater than or equal to low. The default value is 0.

**high**

[float or array_like of floats] Upper boundary of the output interval. All values generated will be less than high. The default value is 1.0.

**size**

[int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `low` and `high` are both scalars. Otherwise, `np.broadcast(low, high).size` samples are drawn.

**Returns**

**out**

[ndarray or scalar] Drawn samples from the parameterized uniform distribution.

**See also:**

***integers***

Discrete uniform distribution, yielding integers.

***random***

Floats uniformly distributed over `[0, 1)`.

## Notes

The probability density function of the uniform distribution is

$$p(x) = \frac{1}{b - a}$$

anywhere within the interval `[a, b)`, and zero elsewhere.

When `high` == `low`, values of `low` will be returned. If `high` < `low`, the results are officially undefined and may eventually raise an error, i.e. do not rely on this function to behave when passed arguments satisfying that inequality condition.

## Examples

Draw samples from the distribution:

```
>>> s = np.random.default_rng().uniform(-1,0,1000)
```

All values are within the given interval:

```
>>> np.all(s >= -1)
True
>>> np.all(s < 0)
True
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 15, density=True)
>>> plt.plot(bins, np.ones_like(bins), linewidth=2, color='r')
>>> plt.show()
```

method

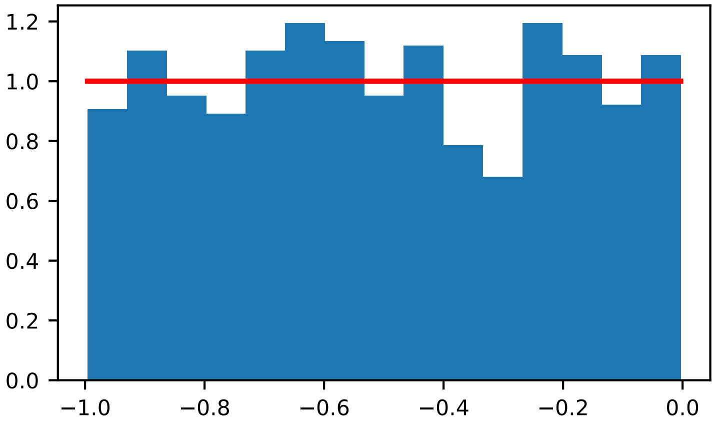
`Generator.`**`vonmises`**(*mu*, *kappa*, *size=None*)

Draw samples from a von Mises distribution.

Samples are drawn from a von Mises distribution with specified mode (mu) and dispersion (kappa), on the interval [-pi, pi].

The von Mises distribution (also known as the circular normal distribution) is a continuous probability distribution on the unit circle. It may be thought of as the circular analogue of the normal distribution.

**Parameters**



**mu**

[float or array_like of floats] Mode ("center") of the distribution.

**kappa**

[float or array_like of floats] Dispersion of the distribution, has to be >=0.

**size**

[int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `mu` and `kappa` are both scalars. Otherwise, `np.broadcast(mu, kappa).size` samples are drawn.

**Returns**

**out**

[ndarray or scalar] Drawn samples from the parameterized von Mises distribution.

**See also:**

`scipy.stats.vonmises`

probability density function, distribution, or cumulative density function, etc.

### Notes

The probability density for the von Mises distribution is

$$p(x) = \frac{e^{\kappa cos(x-\mu)}}{2\pi I_0(\kappa)},$$

where $\mu$ is the mode and $\kappa$ the dispersion, and $I_0(\kappa)$ is the modified Bessel function of order 0.

The von Mises is named for Richard Edler von Mises, who was born in Austria-Hungary, in what is now the Ukraine. He fled to the United States in 1939 and became a professor at Harvard. He worked in probability theory, aerodynamics, fluid mechanics, and philosophy of science.

### References

[1], [2]

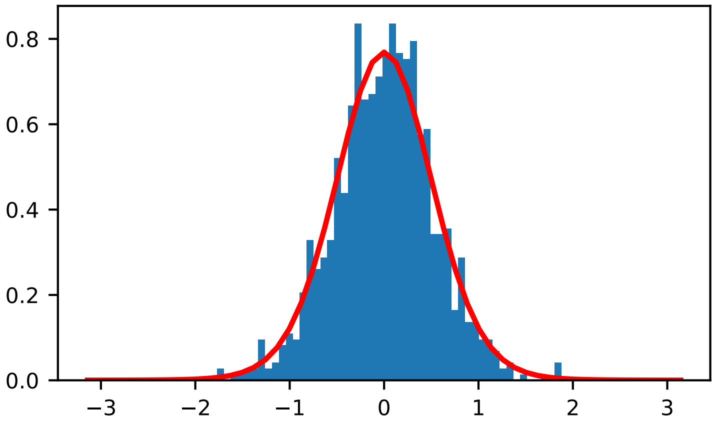
### Examples

Draw samples from the distribution:

```
>>> mu, kappa = 0.0, 4.0 # mean and dispersion
>>> s = np.random.default_rng().vonmises(mu, kappa, 1000)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> from scipy.special import i0
>>> plt.hist(s, 50, density=True)
>>> x = np.linspace(-np.pi, np.pi, num=51)
>>> y = np.exp(kappa*np.cos(x-mu))/(2*np.pi*i0(kappa))
>>> plt.plot(x, y, linewidth=2, color='r')
>>> plt.show()
```



method

`Generator.`**`wald`**(*mean*, *scale*, *size=None*)

Draw samples from a Wald, or inverse Gaussian, distribution.

As the scale approaches infinity, the distribution becomes more like a Gaussian. Some references claim that the Wald is an inverse Gaussian with mean equal to 1, but this is by no means universal.

The inverse Gaussian distribution was first studied in relationship to Brownian motion. In 1956 M.C.K. Tweedie used the name inverse Gaussian because there is an inverse relationship between the time to cover a unit distance and distance covered in unit time.

**Parameters**

**mean**

[float or array_like of floats] Distribution mean, must be > 0.

**scale**

[float or array_like of floats] Scale parameter, must be > 0.

**size**

[int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `mean` and `scale` are both scalars. Otherwise, `np.broadcast(mean, scale).size` samples are drawn.

**Returns**

**out**

[ndarray or scalar] Drawn samples from the parameterized Wald distribution.

## Notes

The probability density function for the Wald distribution is

$$P(x; mean, scale) = \sqrt{\frac{scale}{2\pi x^3}}e^{\frac{-scale(x-mean)^2}{2\cdot mean^2 x}}$$

As noted above the inverse Gaussian distribution first arise from attempts to model Brownian motion. It is also a competitor to the Weibull for use in reliability modeling and modeling stock returns and interest rate processes.

## References

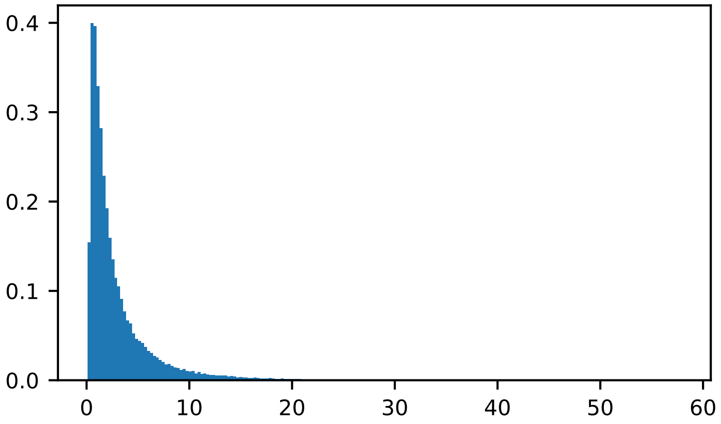
[1], [2], [3]

## Examples

Draw values from the distribution and plot the histogram:

```
>>> import matplotlib.pyplot as plt
>>> h = plt.hist(np.random.default_rng().wald(3, 2, 100000), bins=200,␣
↪density=True)
>>> plt.show()
```

method



`Generator.`**`weibull`**`(`*a*, *size=None*`)`

Draw samples from a Weibull distribution.

Draw samples from a 1-parameter Weibull distribution with the given shape parameter *a*.

$$X = (-ln(U))^{1/a}$$

Here, U is drawn from the uniform distribution over (0,1].

The more common 2-parameter Weibull, including a scale parameter $\lambda$ is just $X = \lambda(-ln(U))^{1/a}$.

**Parameters**

**a**

[float or array_like of floats] Shape parameter of the distribution. Must be nonnegative.

**size**

[int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `a` is a scalar. Otherwise, `np.array(a).size` samples are drawn.

**Returns**

**out**

[ndarray or scalar] Drawn samples from the parameterized Weibull distribution.

**See also:**

`scipy.stats.weibull_max`, `scipy.stats.weibull_min`, `scipy.stats.genextreme`, *`gumbel`*

### Notes

The Weibull (or Type III asymptotic extreme value distribution for smallest values, SEV Type III, or Rosin-Rammler distribution) is one of a class of Generalized Extreme Value (GEV) distributions used in modeling extreme value problems. This class includes the Gumbel and Frechet distributions.

The probability density for the Weibull distribution is

$$p(x) = \frac{a}{\lambda}(\frac{x}{\lambda})^{a-1}e^{-(x/\lambda)^a},$$

where $a$ is the shape and $\lambda$ the scale.

The function has its peak (the mode) at $\lambda(\frac{a-1}{a})^{1/a}$.

When `a = 1`, the Weibull distribution reduces to the exponential distribution.

### References

[1], [2], [3]

### Examples

Draw samples from the distribution:

```
>>> rng = np.random.default_rng()
>>> a = 5. # shape
>>> s = rng.weibull(a, 1000)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> x = np.arange(1,100.)/50.
>>> def weib(x,n,a):
...     return (a / n) * (x / n)**(a - 1) * np.exp(-(x / n)**a)
```

```
>>> count, bins, ignored = plt.hist(rng.weibull(5.,1000))
>>> x = np.arange(1,100.)/50.
>>> scale = count.max()/weib(x, 1., 5.).max()
>>> plt.plot(x, weib(x, 1., 5.)*scale)
>>> plt.show()
```

method

`Generator.`**`zipf`**(*a*, *size=None*)

Draw samples from a Zipf distribution.

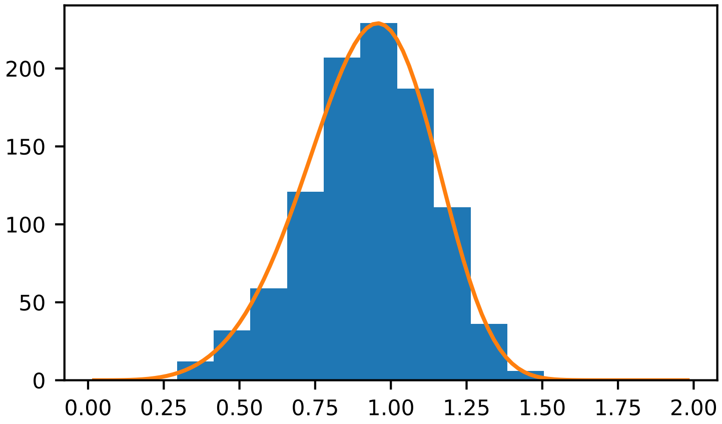
Samples are drawn from a Zipf distribution with specified parameter *a* > 1.

The Zipf distribution (also known as the zeta distribution) is a continuous probability distribution that satisfies Zipf's law: the frequency of an item is inversely proportional to its rank in a frequency table.

**Parameters**

**a**

[float or array_like of floats] Distribution parameter. Must be greater than 1.



**size**

[int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `a` is a scalar. Otherwise, `np.array(a).size` samples are drawn.

**Returns**

**out**

[ndarray or scalar] Drawn samples from the parameterized Zipf distribution.

**See also:**

**`scipy.stats.zipf`**

probability density function, distribution, or cumulative density function, etc.

## Notes

The probability density for the Zipf distribution is

$$p(x) = \frac{x^{-a}}{\zeta(a)},$$

where $\zeta$ is the Riemann Zeta function.

It is named for the American linguist George Kingsley Zipf, who noted that the frequency of any word in a sample of a language is inversely proportional to its rank in the frequency table.

### References

[1]

### Examples

Draw samples from the distribution:

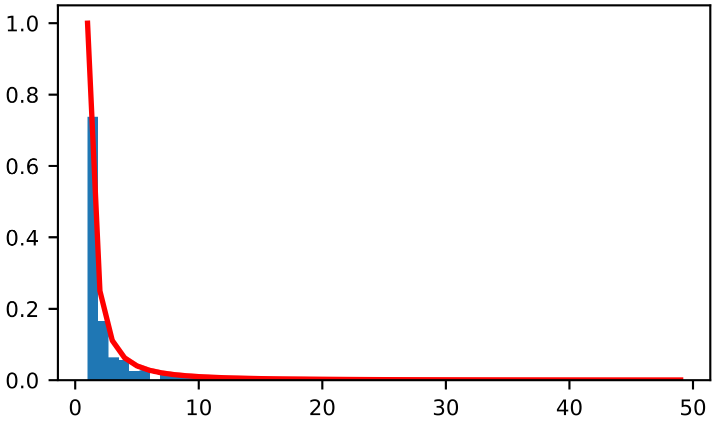
```
>>> a = 2. # parameter
>>> s = np.random.default_rng().zipf(a, 1000)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> from scipy import special
```

Truncate s values at 50 so plot is interesting:

```
>>> count, bins, ignored = plt.hist(s[s<50],
...         50, density=True)
>>> x = np.arange(1., 50.)
>>> y = x**(-a) / special.zetac(a)
>>> plt.plot(x, y/max(y), linewidth=2, color='r')
>>> plt.show()
```



### Legacy Random Generation

The `RandomState` provides access to legacy generators. This generator is considered frozen and will have no further improvements. It is guaranteed to produce the same values as the final point release of NumPy v1.16. These all depend on Box-Muller normals or inverse CDF exponentials or gammas. This class should only be used if it is essential to have randoms that are identical to what would have been produced by previous versions of NumPy.

`RandomState` adds additional information to the state which is required when using Box-Muller normals since these are produced in pairs. It is important to use `RandomState.get_state`, and not the underlying bit generators *state*, when accessing the state so that these extra values are saved.

Although we provide the `MT19937` BitGenerator for use independent of `RandomState`, note that its default seeding uses `SeedSequence` rather than the legacy seeding algorithm. `RandomState` will use the legacy seeding algorithm. The methods to use the legacy seeding algorithm are currently private as the main reason to use them is just to implement `RandomState`. However, one can reset the state of `MT19937` using the state of the `RandomState`:

```
from numpy.random import MT19937
from numpy.random import RandomState

rs = RandomState(12345)
mt19937 = MT19937()
mt19937.state = rs.get_state()
rs2 = RandomState(mt19937)

# Same output
rs.standard_normal()
rs2.standard_normal()

rs.random()
rs2.random()

rs.standard_exponential()
rs2.standard_exponential()
```

**class** `numpy.random.`**`RandomState`**(*seed=None*)

Container for the slow Mersenne Twister pseudo-random number generator. Consider using a different BitGenerator with the Generator container instead.

`RandomState` and `Generator` expose a number of methods for generating random numbers drawn from a variety of probability distributions. In addition to the distribution-specific arguments, each method takes a keyword argument *size* that defaults to `None`. If *size* is `None`, then a single value is generated and returned. If *size* is an integer, then a 1-D array filled with generated values is returned. If *size* is a tuple, then an array with that shape is filled and returned.

**Compatibility Guarantee**

A fixed bit generator using a fixed seed and a fixed series of calls to 'RandomState' methods using the same parameters will always produce the same results up to roundoff error except when the values were incorrect. `RandomState` is effectively frozen and will only receive updates that are required by changes in the the internals of Numpy. More substantial changes, including algorithmic improvements, are reserved for `Generator`.

**Parameters**

**seed**

[{None, int, array_like, BitGenerator}, optional] Random seed used to initialize the pseudo-random number generator or an instantized BitGenerator. If an integer or array, used as a seed for the MT19937 BitGenerator. Values can be any integer between 0 and 2**32 - 1 inclusive, an array (or other sequence) of such integers, or `None` (the default). If `seed` is

None, then the *MT19937* BitGenerator is initialized by reading data from `/dev/urandom` (or the Windows analogue) if available or seed from the clock otherwise.

**See also:**

*Generator*, *MT19937*, *numpy.random.BitGenerator*

### Notes

The Python stdlib module "random" also contains a Mersenne Twister pseudo-random number generator with a number of methods that are similar to the ones available in *RandomState*. *RandomState*, besides being NumPy-aware, has the advantage that it provides a much larger number of probability distributions to choose from.

## Seeding and State

| | |
|---|---|
| *get_state*() | Return a tuple representing the internal state of the generator. |
| *set_state*(state) | Set the internal state of the generator from a tuple. |
| *seed*(self[, seed]) | Reseed a legacy MT19937 BitGenerator |

method

`RandomState.`**`get_state`**`()`

Return a tuple representing the internal state of the generator.

For more details, see *set_state*.

**Parameters**

**legacy**

[bool, optional] Flag indicating to return a legacy tuple state when the BitGenerator is MT19937, instead of a dict.

**Returns**

**out**

[{tuple(str, ndarray of 624 uints, int, int, float), dict}] The returned tuple has the following items:

1. the string 'MT19937'.
2. a 1-D array of 624 unsigned integer keys.
3. an integer `pos`.
4. an integer `has_gauss`.
5. a float `cached_gaussian`.

If *legacy* is False, or the BitGenerator is not MT19937, then state is returned as a dictionary.

**See also:**

*set_state*

### Notes

*`set_state`* and *`get_state`* are not needed to work with any of the random distributions in NumPy. If the internal state is manually altered, the user should know exactly what he/she is doing.

method

`RandomState.`**`set_state`**`(`*state*`)`

Set the internal state of the generator from a tuple.

For use if one has reason to manually (re-)set the internal state of the bit generator used by the RandomState instance. By default, RandomState uses the "Mersenne Twister"[1] pseudo-random number generating algorithm.

**Parameters**

**state**

[{tuple(str, ndarray of 624 uints, int, int, float), dict}] The *state* tuple has the following items:

1. the string 'MT19937', specifying the Mersenne Twister algorithm.
2. a 1-D array of 624 unsigned integers `keys`.
3. an integer `pos`.
4. an integer `has_gauss`.
5. a float `cached_gaussian`.

If state is a dictionary, it is directly set using the BitGenerators *state* property.

**Returns**

**out**

[None] Returns 'None' on success.

**See also:**

*`get_state`*

### Notes

*`set_state`* and *`get_state`* are not needed to work with any of the random distributions in NumPy. If the internal state is manually altered, the user should know exactly what he/she is doing.

For backwards compatibility, the form (str, array of 624 uints, int) is also accepted although it is missing some information about the cached Gaussian value: `state = ('MT19937', keys, pos)`.

### References

[1]

method

`RandomState.`**`seed`**`(`*self*, *seed=None*`)`

Reseed a legacy MT19937 BitGenerator

### Notes

This is a convenience, legacy function.

The best practice is to **not** reseed a BitGenerator, rather to recreate a new one. This method is here for legacy reasons. This example demonstrates best practice.

```
>>> from numpy.random import MT19937
>>> from numpy.random import RandomState, SeedSequence
>>> rs = RandomState(MT19937(SeedSequence(123456789)))
# Later, you want to restart the stream
>>> rs = RandomState(MT19937(SeedSequence(987654321)))
```

## Simple random data

| | |
|---|---|
| `rand`(d0, d1, …, dn) | Random values in a given shape. |
| `randn`(d0, d1, …, dn) | Return a sample (or samples) from the "standard normal" distribution. |
| `randint`(low[, high, size, dtype]) | Return random integers from *low* (inclusive) to *high* (exclusive). |
| `random_integers`(low[, high, size]) | Random integers of type *np.int_* between *low* and *high*, inclusive. |
| `random_sample`([size]) | Return random floats in the half-open interval [0.0, 1.0). |
| `choice`(a[, size, replace, p]) | Generates a random sample from a given 1-D array |
| `bytes`(length) | Return random bytes. |

method

`RandomState.`**`rand`**`(`*d0, d1, ..., dn*`)`

Random values in a given shape.

**Note:** This is a convenience function for users porting code from Matlab, and wraps `random_sample`. That function takes a tuple to specify the size of the output, which is consistent with other NumPy functions like `numpy.zeros` and `numpy.ones`.

Create an array of the given shape and populate it with random samples from a uniform distribution over `[0, 1)`.

**Parameters**

**d0, d1, …, dn**

[int, optional] The dimensions of the returned array, must be non-negative. If no argument is given a single Python float is returned.

**Returns**

**out**

[ndarray, shape `(d0, d1, ..., dn)`] Random values.

**See also:**

`random`

### Examples

```
>>> np.random.rand(3,2)
array([[ 0.14022471,  0.96360618],  #random
       [ 0.37601032,  0.25528411],  #random
       [ 0.49313049,  0.94909878]]) #random
```

method

`RandomState.`**`randn`**`(`*d0*, *d1*, *...*, *dn*`)`

Return a sample (or samples) from the "standard normal" distribution.

---

**Note:** This is a convenience function for users porting code from Matlab, and wraps *`standard_normal`*. That function takes a tuple to specify the size of the output, which is consistent with other NumPy functions like *`numpy.zeros`* and *`numpy.ones`*.

---

**Note:** New code should use the `standard_normal` method of a `default_rng()` instance instead; see *random-quick-start*.

---

If positive int_like arguments are provided, *`randn`* generates an array of shape `(d0, d1, ..., dn)`, filled with random floats sampled from a univariate "normal" (Gaussian) distribution of mean 0 and variance 1. A single float randomly sampled from the distribution is returned if no argument is provided.

**Parameters**

**d0, d1, …, dn**

[int, optional] The dimensions of the returned array, must be non-negative. If no argument is given a single Python float is returned.

**Returns**

**Z**

[ndarray or float] A `(d0, d1, ..., dn)`-shaped array of floating-point samples from the standard normal distribution, or a single such float if no parameters were supplied.

**See also:**

***`standard_normal`***

Similar, but takes a tuple as its argument.

***`normal`***

Also accepts mu and sigma arguments.

***`Generator.standard_normal`***

which should be used for new code.

## Notes

For random samples from $N(\mu, \sigma^2)$, use:

`sigma * np.random.randn(...) + mu`

## Examples

```
>>> np.random.randn()
2.1923875335537315  # random
```

Two-by-four array of samples from N(3, 6.25):

```
>>> 3 + 2.5 * np.random.randn(2, 4)
array([[-4.49401501,  4.00950034, -1.81814867,  7.29718677],   # random
       [ 0.39924804,  4.68456316,  4.99394529,  4.84057254]])  # random
```

method

`RandomState.`**`randint`**(*low*, *high=None*, *size=None*, *dtype=int*)

Return random integers from *low* (inclusive) to *high* (exclusive).

Return random integers from the "discrete uniform" distribution of the specified dtype in the "half-open" interval [*low*, *high*). If *high* is None (the default), then results are from [0, *low*).

**Note:** New code should use the `integers` method of a `default_rng()` instance instead; see *random-quick-start*.

**Parameters**

**low**

[int or array-like of ints] Lowest (signed) integers to be drawn from the distribution (unless `high=None`, in which case this parameter is one above the *highest* such integer).

**high**

[int or array-like of ints, optional] If provided, one above the largest (signed) integer to be drawn from the distribution (see above for behavior if `high=None`). If array-like, must contain integer values

**size**

[int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. Default is None, in which case a single value is returned.

**dtype**

[dtype, optional] Desired dtype of the result. Byteorder must be native. The default value is int.

New in version 1.11.0.

**Returns**

**out**

[int or ndarray of ints] *size*-shaped array of random integers from the appropriate distribution, or a single such random int if *size* not provided.

**See also:**

*`random_integers`*

similar to *`randint`*, only for the closed interval [*low*, *high*], and 1 is the lowest value if *high* is omitted.

*`Generator.integers`*

which should be used for new code.

## Examples

```
>>> np.random.randint(2, size=10)
array([1, 0, 0, 0, 1, 1, 0, 0, 1, 0]) # random
>>> np.random.randint(1, size=10)
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
```

Generate a 2 x 4 array of ints between 0 and 4, inclusive:

```
>>> np.random.randint(5, size=(2, 4))
array([[4, 0, 2, 1], # random
       [3, 2, 2, 0]])
```

Generate a 1 x 3 array with 3 different upper bounds

```
>>> np.random.randint(1, [3, 5, 10])
array([2, 2, 9]) # random
```

Generate a 1 by 3 array with 3 different lower bounds

```
>>> np.random.randint([1, 5, 7], 10)
array([9, 8, 7]) # random
```

Generate a 2 by 4 array using broadcasting with dtype of uint8

```
>>> np.random.randint([1, 3, 5, 7], [[10], [20]], dtype=np.uint8)
array([[ 8,  6,  9,  7], # random
       [ 1, 16,  9, 12]], dtype=uint8)
```

method

`RandomState.`**`random_integers`**(*low*, *high=None*, *size=None*)

Random integers of type *np.int_* between *low* and *high*, inclusive.

Return random integers of type *np.int_* from the “discrete uniform” distribution in the closed interval [*low*, *high*]. If *high* is None (the default), then results are from [1, *low*]. The *np.int_* type translates to the C long integer type and its precision is platform dependent.

This function has been deprecated. Use randint instead.

Deprecated since version 1.11.0.

**Parameters**

**low**

[int] Lowest (signed) integer to be drawn from the distribution (unless `high=None`, in which case this parameter is the *highest* such integer).

**high**

[int, optional] If provided, the largest (signed) integer to be drawn from the distribution (see above for behavior if `high=None`).

**size**

[int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. Default is None, in which case a single value is returned.

**Returns**

**out**

[int or ndarray of ints] *size*-shaped array of random integers from the appropriate distribution, or a single such random int if *size* not provided.

**See also:**

***`randint`***

Similar to *`random_integers`*, only for the half-open interval [*low*, *high*), and 0 is the lowest value if *high* is omitted.

## Notes

To sample from N evenly spaced floating-point numbers between a and b, use:

```
a + (b - a) * (np.random.random_integers(N) - 1) / (N - 1.)
```

## Examples

```
>>> np.random.random_integers(5)
4 # random
>>> type(np.random.random_integers(5))
<class 'numpy.int64'>
>>> np.random.random_integers(5, size=(3,2))
array([[5, 4], # random
       [3, 3],
       [4, 5]])
```

Choose five random numbers from the set of five evenly-spaced numbers between 0 and 2.5, inclusive (*i.e.*, from the set $0, 5/8, 10/8, 15/8, 20/8$):

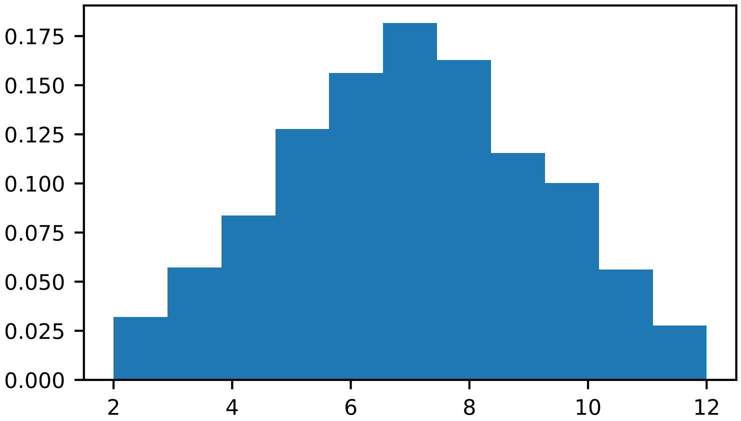
```
>>> 2.5 * (np.random.random_integers(5, size=(5,)) - 1) / 4.
array([ 0.625,  1.25 ,  0.625,  0.625,  2.5  ]) # random
```

Roll two six sided dice 1000 times and sum the results:

```
>>> d1 = np.random.random_integers(1, 6, 1000)
>>> d2 = np.random.random_integers(1, 6, 1000)
>>> dsums = d1 + d2
```

Display results as a histogram:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(dsums, 11, density=True)
>>> plt.show()
```



method

`RandomState.`**`random_sample`**`(`*size=None*`)`

Return random floats in the half-open interval [0.0, 1.0).

Results are from the "continuous uniform" distribution over the stated interval. To sample $Unif[a, b), b > a$ multiply the output of *`random_sample`* by *(b-a)* and add *a*:

```
(b - a) * random_sample() + a
```

---

**Note:** New code should use the `random` method of a `default_rng()` instance instead; see *random-quick-start*.

---

**Parameters**

**size**

[int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. Default is None, in which case a single value is returned.

**Returns**

**out**

[float or ndarray of floats] Array of random floats of shape *size* (unless `size=None`, in which case a single float is returned).

**See also:**

*Generator.random*

which should be used for new code.

### Examples

```
>>> np.random.random_sample()
0.47108547995356098 # random
>>> type(np.random.random_sample())
<class 'float'>
>>> np.random.random_sample((5,))
array([ 0.30220482,  0.86820401,  0.1654503 ,  0.11659149,  0.54323428]) # random
```

Three-by-two array of random numbers from [-5, 0):

```
>>> 5 * np.random.random_sample((3, 2)) - 5
array([[-3.99149989, -0.52338984], # random
       [-2.99091858, -0.79479508],
       [-1.23204345, -1.75224494]])
```

method

`RandomState.`**`choice`**(*a*, *size=None*, *replace=True*, *p=None*)

Generates a random sample from a given 1-D array

New in version 1.7.0.

**Note:** New code should use the `choice` method of a `default_rng()` instance instead; see *random-quick-start*.

**Parameters**

**a**

[1-D array-like or int] If an ndarray, a random sample is generated from its elements. If an int, the random sample is generated as if a were np.arange(a)

**size**

[int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. Default is None, in which case a single value is returned.

**replace**

[boolean, optional] Whether the sample is with or without replacement

**p**

[1-D array-like, optional] The probabilities associated with each entry in a. If not given the sample assumes a uniform distribution over all entries in a.

**Returns**

**samples**

[single item or ndarray] The generated random samples

**Raises**

**ValueError**

If a is an int and less than zero, if a or p are not 1-dimensional, if a is an array-like of size 0, if p is not a vector of probabilities, if a and p have different lengths, or if replace=False and the sample size is greater than the population size

**See also:**

*`randint`*, *`shuffle`*, *`permutation`*

***`Generator.choice`***

which should be used in new code

## Notes

Sampling random rows from a 2-D array is not possible with this function, but is possible with *`Generator.choice`* through its `axis` keyword.

## Examples

Generate a uniform random sample from np.arange(5) of size 3:

```
>>> np.random.choice(5, 3)
array([0, 3, 4]) # random
>>> #This is equivalent to np.random.randint(0,5,3)
```

Generate a non-uniform random sample from np.arange(5) of size 3:

```
>>> np.random.choice(5, 3, p=[0.1, 0, 0.3, 0.6, 0])
array([3, 3, 0]) # random
```

Generate a uniform random sample from np.arange(5) of size 3 without replacement:

```
>>> np.random.choice(5, 3, replace=False)
array([3,1,0]) # random
>>> #This is equivalent to np.random.permutation(np.arange(5))[:3]
```

Generate a non-uniform random sample from np.arange(5) of size 3 without replacement:

```
>>> np.random.choice(5, 3, replace=False, p=[0.1, 0, 0.3, 0.6, 0])
array([2, 3, 0]) # random
```

Any of the above can be repeated with an arbitrary array-like instead of just integers. For instance:

```
>>> aa_milne_arr = ['pooh', 'rabbit', 'piglet', 'Christopher']
>>> np.random.choice(aa_milne_arr, 5, p=[0.5, 0.1, 0.1, 0.3])
array(['pooh', 'pooh', 'pooh', 'Christopher', 'piglet'], # random
      dtype='<U11')
```

method

`RandomState.`**`bytes`**`(`*length*`)`

Return random bytes.

---

**Note:** New code should use the `bytes` method of a `default_rng()` instance instead; see *random-quick-start*.

---

**Parameters**

**length**

[int] Number of random bytes.

**Returns**

**out**

[str] String of length *length*.

**See also:**

***Generator.bytes***

which should be used for new code.

### Examples

```
>>> np.random.bytes(10)
' eh\x85\x022SZ\xbf\xa4' #random
```

## Permutations

| | |
|---|---|
| *shuffle*(x) | Modify a sequence in-place by shuffling its contents. |
| *permutation*(x) | Randomly permute a sequence, or return a permuted range. |

method

`RandomState.`**`shuffle`**`(`*x*`)`

Modify a sequence in-place by shuffling its contents.

This function only shuffles the array along the first axis of a multi-dimensional array. The order of sub-arrays is changed but their contents remains the same.

**Note:** New code should use the `shuffle` method of a `default_rng()` instance instead; see *random-quick-start*.

**Parameters**

**x**

[array_like] The array or list to be shuffled.

**Returns**

**None**

**See also:**

***Generator.shuffle***

which should be used for new code.

### Examples

```
>>> arr = np.arange(10)
>>> np.random.shuffle(arr)
>>> arr
[1 7 5 2 9 4 3 6 0 8] # random
```

Multi-dimensional arrays are only shuffled along the first axis:

```
>>> arr = np.arange(9).reshape((3, 3))
>>> np.random.shuffle(arr)
>>> arr
array([[3, 4, 5], # random
       [6, 7, 8],
       [0, 1, 2]])
```

method

`RandomState.`**`permutation`**`(`*x*`)`

Randomly permute a sequence, or return a permuted range.

If *x* is a multi-dimensional array, it is only shuffled along its first index.

---

**Note:** New code should use the `permutation` method of a `default_rng()` instance instead; see *random-quick-start*.

---

**Parameters**

**x**

[int or array_like] If *x* is an integer, randomly permute `np.arange(x)`. If *x* is an array, make a copy and shuffle the elements randomly.

**Returns**

**out**

[ndarray] Permuted sequence or array range.

**See also:**

*`Generator.permutation`*

which should be used for new code.

### Examples

```
>>> np.random.permutation(10)
array([1, 7, 4, 3, 0, 9, 2, 5, 8, 6]) # random
```

```
>>> np.random.permutation([1, 4, 9, 12, 15])
array([15,  1,  9,  4, 12]) # random
```

```
>>> arr = np.arange(9).reshape((3, 3))
>>> np.random.permutation(arr)
array([[6, 7, 8], # random
       [0, 1, 2],
       [3, 4, 5]])
```

## Distributions

| | |
|---|---|
| `beta`(a, b[, size]) | Draw samples from a Beta distribution. |
| `binomial`(n, p[, size]) | Draw samples from a binomial distribution. |
| `chisquare`(df[, size]) | Draw samples from a chi-square distribution. |
| `dirichlet`(alpha[, size]) | Draw samples from the Dirichlet distribution. |
| `exponential`([scale, size]) | Draw samples from an exponential distribution. |
| `f`(dfnum, dfden[, size]) | Draw samples from an F distribution. |
| `gamma`(shape[, scale, size]) | Draw samples from a Gamma distribution. |
| `geometric`(p[, size]) | Draw samples from the geometric distribution. |
| `gumbel`([loc, scale, size]) | Draw samples from a Gumbel distribution. |
| `hypergeometric`(ngood, nbad, nsample[, size]) | Draw samples from a Hypergeometric distribution. |
| `laplace`([loc, scale, size]) | Draw samples from the Laplace or double exponential distribution with specified location (or mean) and scale (decay). |
| `logistic`([loc, scale, size]) | Draw samples from a logistic distribution. |
| `lognormal`([mean, sigma, size]) | Draw samples from a log-normal distribution. |
| `logseries`(p[, size]) | Draw samples from a logarithmic series distribution. |
| `multinomial`(n, pvals[, size]) | Draw samples from a multinomial distribution. |
| `multivariate_normal`(mean, cov[, size, …]) | Draw random samples from a multivariate normal distribution. |
| `negative_binomial`(n, p[, size]) | Draw samples from a negative binomial distribution. |
| `noncentral_chisquare`(df, nonc[, size]) | Draw samples from a noncentral chi-square distribution. |
| `noncentral_f`(dfnum, dfden, nonc[, size]) | Draw samples from the noncentral F distribution. |
| `normal`([loc, scale, size]) | Draw random samples from a normal (Gaussian) distribution. |
| `pareto`(a[, size]) | Draw samples from a Pareto II or Lomax distribution with specified shape. |
| `poisson`([lam, size]) | Draw samples from a Poisson distribution. |
| `power`(a[, size]) | Draws samples in [0, 1] from a power distribution with positive exponent a - 1. |
| `rayleigh`([scale, size]) | Draw samples from a Rayleigh distribution. |
| `standard_cauchy`([size]) | Draw samples from a standard Cauchy distribution with mode = 0. |
| `standard_exponential`([size]) | Draw samples from the standard exponential distribution. |
| `standard_gamma`(shape[, size]) | Draw samples from a standard Gamma distribution. |

Continued on next page

Table 150 – continued from previous page

| | |
|---|---|
| *standard_normal*([size]) | Draw samples from a standard Normal distribution (mean=0, stdev=1). |
| *standard_t*(df[, size]) | Draw samples from a standard Student's t distribution with *df* degrees of freedom. |
| *triangular*(left, mode, right[, size]) | Draw samples from the triangular distribution over the interval `[left, right]`. |
| *uniform*([low, high, size]) | Draw samples from a uniform distribution. |
| *vonmises*(mu, kappa[, size]) | Draw samples from a von Mises distribution. |
| *wald*(mean, scale[, size]) | Draw samples from a Wald, or inverse Gaussian, distribution. |
| *weibull*(a[, size]) | Draw samples from a Weibull distribution. |
| *zipf*(a[, size]) | Draw samples from a Zipf distribution. |

method

`RandomState.`**`beta`**(*a*, *b*, *size=None*)

Draw samples from a Beta distribution.

The Beta distribution is a special case of the Dirichlet distribution, and is related to the Gamma distribution. It has the probability distribution function

$$f(x; a,b) = \frac{1}{B(\alpha, \beta)} x^{\alpha - 1} (1 - x)^{\beta - 1},$$

where the normalization, B, is the beta function,

$$B(\alpha, \beta) = \int_0^1 t^{\alpha - 1} (1 - t)^{\beta - 1} dt.$$

It is often seen in Bayesian inference and order statistics.

**Note:** New code should use the `beta` method of a `default_rng()` instance instead; see *random-quick-start*.

**Parameters**

**a**

[float or array_like of floats] Alpha, positive (>0).

**b**

[float or array_like of floats] Beta, positive (>0).

**size**

[int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `a` and `b` are both scalars. Otherwise, `np.broadcast(a, b).size` samples are drawn.

**Returns**

**out**

[ndarray or scalar] Drawn samples from the parameterized beta distribution.

**See also:**

**`Generator.beta`**

which should be used for new code.

method

`RandomState.`**`binomial`**(*n*, *p*, *size=None*)

Draw samples from a binomial distribution.

Samples are drawn from a binomial distribution with specified parameters, n trials and p probability of success where n an integer >= 0 and p is in the interval [0,1]. (n may be input as a float, but it is truncated to an integer in use)

---

**Note:** New code should use the `binomial` method of a `default_rng()` instance instead; see *random-quick-start*.

---

**Parameters**

**n**

[int or array_like of ints] Parameter of the distribution, >= 0. Floats are also accepted, but they will be truncated to integers.

**p**

[float or array_like of floats] Parameter of the distribution, >= 0 and <=1.

**size**

[int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `n` and `p` are both scalars. Otherwise, `np.broadcast(n, p).size` samples are drawn.

**Returns**

**out**

[ndarray or scalar] Drawn samples from the parameterized binomial distribution, where each sample is equal to the number of successes over the n trials.

**See also:**

**`scipy.stats.binom`**

probability density function, distribution or cumulative density function, etc.

**`Generator.binomial`**

which should be used for new code.

## Notes

The probability density for the binomial distribution is

$$P(N) = \binom{n}{N}p^N(1-p)^{n-N},$$

where $n$ is the number of trials, $p$ is the probability of success, and $N$ is the number of successes.

When estimating the standard error of a proportion in a population by using a random sample, the normal distribution works well unless the product p*n <=5, where p = population proportion estimate, and n = number of samples, in which case the binomial distribution is used instead. For example, a sample of 15 people shows 4 who are left handed, and 11 who are right handed. Then p = 4/15 = 27%. 0.27*15 = 4, so the binomial distribution should be used in this case.

## References

[1], [2], [3], [4], [5]

## Examples

Draw samples from the distribution:

```
>>> n, p = 10, .5  # number of trials, probability of each trial
>>> s = np.random.binomial(n, p, 1000)
# result of flipping a coin 10 times, tested 1000 times.
```

A real world example. A company drills 9 wild-cat oil exploration wells, each with an estimated probability of success of 0.1. All nine wells fail. What is the probability of that happening?

Let's do 20,000 trials of the model, and count the number that generate zero positive results.

```
>>> sum(np.random.binomial(9, 0.1, 20000) == 0)/20000.
# answer = 0.38885, or 38%.
```

method

`RandomState.`**`chisquare`**(*df*, *size=None*)

Draw samples from a chi-square distribution.

When *df* independent random variables, each with standard normal distributions (mean 0, variance 1), are squared and summed, the resulting distribution is chi-square (see Notes). This distribution is often used in hypothesis testing.

**Note:** New code should use the `chisquare` method of a `default_rng()` instance instead; see *random-quick-start*.

### Parameters

**df**

[float or array_like of floats] Number of degrees of freedom, must be > 0.

**size**

[int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `df` is a scalar. Otherwise, `np.array(df).size` samples are drawn.

**Returns**

**out**

[ndarray or scalar] Drawn samples from the parameterized chi-square distribution.

**Raises**

**ValueError**

When *df* <= 0 or when an inappropriate *size* (e.g. `size=-1`) is given.

**See also:**

***`Generator.chisquare`***

which should be used for new code.

## Notes

The variable obtained by summing the squares of *df* independent, standard normally distributed random variables:

$$Q = \sum_{i=0}^{\mathtt{df}} X_i^2$$

is chi-square distributed, denoted

$$Q \sim \chi^2_k.$$

The probability density function of the chi-squared distribution is

$$p(x) = \frac{(1/2)^{k/2}}{\Gamma(k/2)} x^{k/2 - 1} e^{-x/2},$$

where $\Gamma$ is the gamma function,

$$\Gamma(x) = \int_0^{-\infty} t^{x - 1} e^{-t} dt.$$

## References

[1]

## Examples

```
>>> np.random.chisquare(2,4)
array([ 1.89920014,  9.00867716,  3.13710533,  5.62318272]) # random
```

method

`RandomState.`**`dirichlet`**`(`*alpha*, *size=None*`)`

Draw samples from the Dirichlet distribution.

Draw *size* samples of dimension k from a Dirichlet distribution. A Dirichlet-distributed random variable can be seen as a multivariate generalization of a Beta distribution. The Dirichlet distribution is a conjugate prior of a multinomial distribution in Bayesian inference.

**Note:** New code should use the `dirichlet` method of a `default_rng()` instance instead; see *random-quick-start*.

**Parameters**

**alpha**

[sequence of floats, length k] Parameter of the distribution (length `k` for sample of length `k`).

**size**

[int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n)`, then `m * n * k` samples are drawn. Default is None, in which case a vector of length `k` is returned.

**Returns**

**samples**

[ndarray,] The drawn samples, of shape `(size, k)`.

**Raises**

**ValueError**

If any value in `alpha` is less than or equal to zero

**See also:**

*`Generator.dirichlet`*

which should be used for new code.

### Notes

The Dirichlet distribution is a distribution over vectors $x$ that fulfil the conditions $x_i > 0$ and $\sum_{i=1}^{k} x_i = 1$.

The probability density function $p$ of a Dirichlet-distributed random vector $X$ is proportional to

$$p(x) \propto \prod_{i=1}^{k} x_i^{\alpha_i - 1},$$

where $\alpha$ is a vector containing the positive concentration parameters.

The method uses the following property for computation: let $Y$ be a random vector which has components that follow a standard gamma distribution, then $X = \frac{1}{\sum_{i=1}^{k} Y_i} Y$ is Dirichlet-distributed

### References

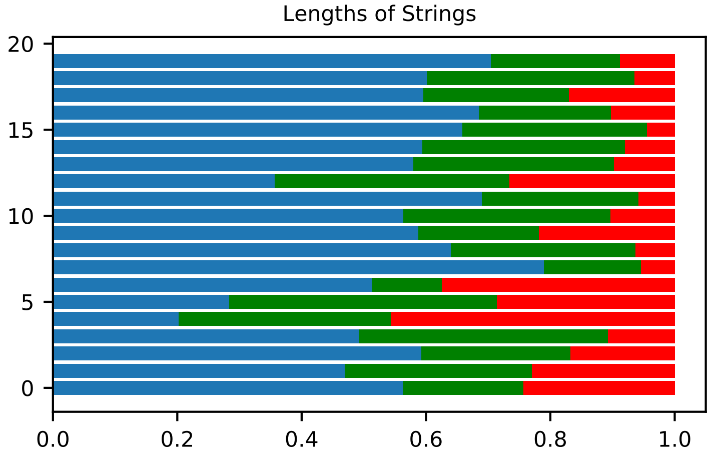
[1], [2]

### Examples

Taking an example cited in Wikipedia, this distribution can be used if one wanted to cut strings (each of initial length 1.0) into K pieces with different lengths, where each piece had, on average, a designated average length, but allowing some variation in the relative sizes of the pieces.

```
>>> s = np.random.dirichlet((10, 5, 3), 20).transpose()
```

```
>>> import matplotlib.pyplot as plt
>>> plt.barh(range(20), s[0])
>>> plt.barh(range(20), s[1], left=s[0], color='g')
>>> plt.barh(range(20), s[2], left=s[0]+s[1], color='r')
>>> plt.title("Lengths of Strings")
```



method

`RandomState.`**`exponential`**(*scale=1.0*, *size=None*)

Draw samples from an exponential distribution.

Its probability density function is

$$f(x; \frac{1}{\beta}) = \frac{1}{\beta} \exp(-\frac{x}{\beta}),$$

for `x > 0` and 0 elsewhere. $\beta$ is the scale parameter, which is the inverse of the rate parameter $\lambda = 1/\beta$. The rate parameter is an alternative, widely used parameterization of the exponential distribution [3].

The exponential distribution is a continuous analogue of the geometric distribution. It describes many common situations, such as the size of raindrops measured over many rainstorms [1], or the time between page requests to Wikipedia [2].

---

**Note:** New code should use the `exponential` method of a `default_rng()` instance instead; see *random-quick-start*.

---

**Parameters**

**scale**

[float or array_like of floats] The scale parameter, $\beta = 1/\lambda$. Must be non-negative.

**size**

[int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `scale` is a scalar. Otherwise, `np.array(scale).size` samples are drawn.

**Returns**

**out**

[ndarray or scalar] Drawn samples from the parameterized exponential distribution.

**See also:**

***`Generator.exponential`***

which should be used for new code.

## References

[1], [2], [3]

method

`RandomState.`**`f`**(*dfnum*, *dfden*, *size=None*)

Draw samples from an F distribution.

Samples are drawn from an F distribution with specified parameters, *dfnum* (degrees of freedom in numerator) and *dfden* (degrees of freedom in denominator), where both parameters must be greater than zero.

The random variate of the F distribution (also known as the Fisher distribution) is a continuous probability distribution that arises in ANOVA tests, and is the ratio of two chi-square variates.

---

**Note:** New code should use the `f` method of a `default_rng()` instance instead; see *random-quick-start*.

---

**Parameters**

**dfnum**

[float or array_like of floats] Degrees of freedom in numerator, must be > 0.

**dfden**

[float or array_like of float] Degrees of freedom in denominator, must be > 0.

**size**

[int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `dfnum` and `dfden` are both scalars. Otherwise, `np.broadcast(dfnum, dfden).size` samples are drawn.

**Returns**

**out**

[ndarray or scalar] Drawn samples from the parameterized Fisher distribution.

**See also:**

**`scipy.stats.f`**

probability density function, distribution or cumulative density function, etc.

***`Generator.f`***

which should be used for new code.

## Notes

The F statistic is used to compare in-group variances to between-group variances. Calculating the distribution depends on the sampling, and so it is a function of the respective degrees of freedom in the problem. The variable *dfnum* is the number of samples minus one, the between-groups degrees of freedom, while *dfden* is the within-groups degrees of freedom, the sum of the number of samples in each group minus the number of groups.

## References

[1], [2]

### Examples

An example from Glantz[1], pp 47-40:

Two groups, children of diabetics (25 people) and children from people without diabetes (25 controls). Fasting blood glucose was measured, case group had a mean value of 86.1, controls had a mean value of 82.2. Standard deviations were 2.09 and 2.49 respectively. Are these data consistent with the null hypothesis that the parents diabetic status does not affect their children's blood glucose levels? Calculating the F statistic from the data gives a value of 36.01.

Draw samples from the distribution:

```
>>> dfnum = 1. # between group degrees of freedom
>>> dfden = 48. # within groups degrees of freedom
>>> s = np.random.f(dfnum, dfden, 1000)
```

The lower bound for the top 1% of the samples is :

```
>>> np.sort(s)[-10]
7.61988120985 # random
```

So there is about a 1% chance that the F statistic will exceed 7.62, the measured value is 36, so the null hypothesis is rejected at the 1% level.

method

`RandomState.`**`gamma`**`(`*shape*, *scale=1.0*, *size=None*`)`

Draw samples from a Gamma distribution.

Samples are drawn from a Gamma distribution with specified parameters, *shape* (sometimes designated "k") and *scale* (sometimes designated "theta"), where both parameters are > 0.

**Note:** New code should use the `gamma` method of a `default_rng()` instance instead; see *random-quick-start*.

**Parameters**

**shape**

[float or array_like of floats] The shape of the gamma distribution. Must be non-negative.

**scale**

[float or array_like of floats, optional] The scale of the gamma distribution. Must be non-negative. Default is equal to 1.

**size**

[int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `shape` and `scale` are both scalars. Otherwise, `np.broadcast(shape, scale).size` samples are drawn.

**Returns**

**out**

[ndarray or scalar] Drawn samples from the parameterized gamma distribution.

**See also:**

**`scipy.stats.gamma`**

probability density function, distribution or cumulative density function, etc.

***`Generator.gamma`***

which should be used for new code.

### Notes

The probability density for the Gamma distribution is

$$p(x) = x^{k-1}\frac{e^{-x/\theta}}{\theta^k\Gamma(k)},$$

where $k$ is the shape and $\theta$ the scale, and $\Gamma$ is the Gamma function.

The Gamma distribution is often used to model the times to failure of electronic components, and arises naturally in processes for which the waiting times between Poisson distributed events are relevant.

### References

[1], [2]

### Examples

Draw samples from the distribution:

```
>>> shape, scale = 2., 2.  # mean=4, std=2*sqrt(2)
>>> s = np.random.gamma(shape, scale, 1000)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> import scipy.special as sps
>>> count, bins, ignored = plt.hist(s, 50, density=True)
>>> y = bins**(shape-1)*(np.exp(-bins/scale) /
...                      (sps.gamma(shape)*scale**shape))
>>> plt.plot(bins, y, linewidth=2, color='r')
>>> plt.show()
```

method

`RandomState.`**`geometric`**(*p*, *size=None*)

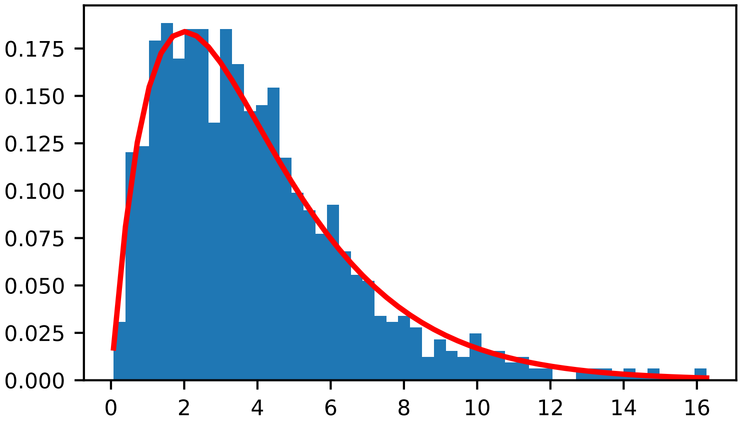
Draw samples from the geometric distribution.

Bernoulli trials are experiments with one of two outcomes: success or failure (an example of such an experiment is flipping a coin). The geometric distribution models the number of trials that must be run in order to achieve success. It is therefore supported on the positive integers, `k = 1, 2, ...`.

The probability mass function of the geometric distribution is

$$f(k) = (1-p)^{k-1}p$$

where $p$ is the probability of success of an individual trial.



---

**Note:** New code should use the `geometric` method of a `default_rng()` instance instead; see *random-quick-start*.

---

**Parameters**

**p**

[float or array_like of floats] The probability of success of an individual trial.

**size**

[int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `p` is a scalar. Otherwise, `np.array(p).size` samples are drawn.

**Returns**

**out**

[ndarray or scalar] Drawn samples from the parameterized geometric distribution.

**See also:**

`Generator.geometric`

which should be used for new code.

### Examples

Draw ten thousand values from the geometric distribution, with the probability of an individual success equal to 0.35:

```
>>> z = np.random.geometric(p=0.35, size=10000)
```

How many trials succeeded after a single run?

```
>>> (z == 1).sum() / 10000.
0.34889999999999999 #random
```

method

`RandomState.`**`gumbel`**(*loc=0.0*, *scale=1.0*, *size=None*)

Draw samples from a Gumbel distribution.

Draw samples from a Gumbel distribution with specified location and scale. For more information on the Gumbel distribution, see Notes and References below.

**Note:** New code should use the `gumbel` method of a `default_rng()` instance instead; see *random-quick-start*.

**Parameters**

**loc**

[float or array_like of floats, optional] The location of the mode of the distribution. Default is 0.

**scale**

[float or array_like of floats, optional] The scale parameter of the distribution. Default is 1. Must be non- negative.

**size**

[int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `loc` and `scale` are both scalars. Otherwise, `np.broadcast(loc, scale).size` samples are drawn.

**Returns**

**out**

[ndarray or scalar] Drawn samples from the parameterized Gumbel distribution.

**See also:**

`scipy.stats.gumbel_l`, `scipy.stats.gumbel_r`, `scipy.stats.genextreme`, *`weibull`*

***`Generator.gumbel`***

which should be used for new code.

### Notes

The Gumbel (or Smallest Extreme Value (SEV) or the Smallest Extreme Value Type I) distribution is one of a class of Generalized Extreme Value (GEV) distributions used in modeling extreme value problems. The Gumbel is a special case of the Extreme Value Type I distribution for maximums from distributions with "exponential-like" tails.

The probability density for the Gumbel distribution is

$$p(x) = \frac{e^{-(x - \mu)/ \beta}}{\beta} e^{ -e^{-(x - \mu)/ \beta}},$$

where $\mu$ is the mode, a location parameter, and $\beta$ is the scale parameter.

The Gumbel (named for German mathematician Emil Julius Gumbel) was used very early in the hydrology literature, for modeling the occurrence of flood events. It is also used for modeling maximum wind speed and rainfall rates. It is a "fat-tailed" distribution - the probability of an event in the tail of the distribution is larger than if one used a Gaussian, hence the surprisingly frequent occurrence of 100-year floods. Floods were initially modeled as a Gaussian process, which underestimated the frequency of extreme events.

It is one of a class of extreme value distributions, the Generalized Extreme Value (GEV) distributions, which also includes the Weibull and Frechet.

The function has a mean of $\mu + 0.57721\beta$ and a variance of $\frac{\pi^2}{6}\beta^2$.

### References

[1], [2]

### Examples

Draw samples from the distribution:

```
>>> mu, beta = 0, 0.1 # location and scale
>>> s = np.random.gumbel(mu, beta, 1000)
```

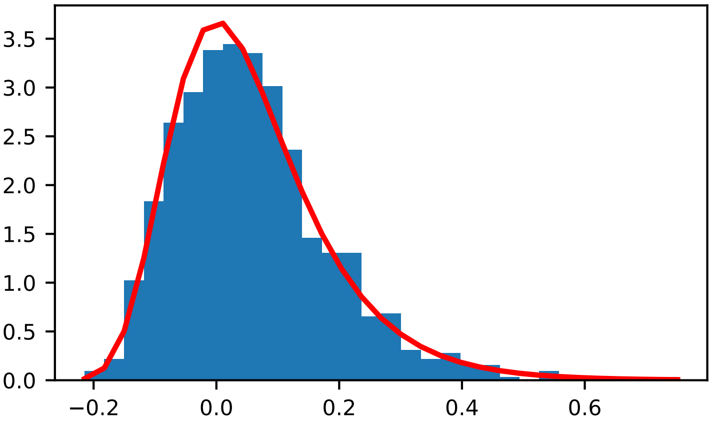
Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 30, density=True)
>>> plt.plot(bins, (1/beta)*np.exp(-(bins - mu)/beta)
...          * np.exp( -np.exp( -(bins - mu) /beta) ),
...          linewidth=2, color='r')
>>> plt.show()
```

Show how an extreme value distribution can arise from a Gaussian process and compare to a Gaussian:

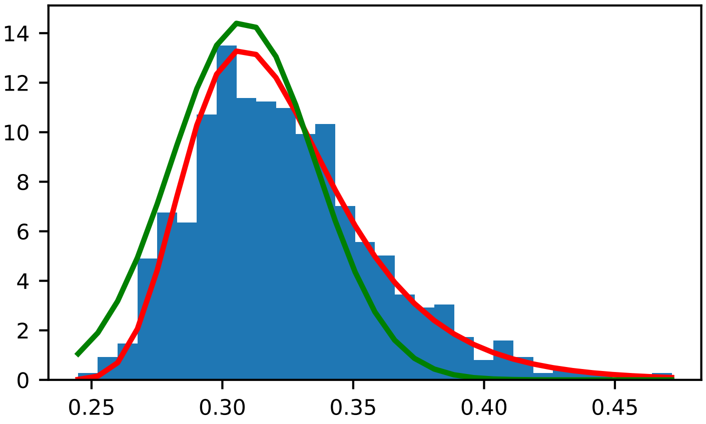
```
>>> means = []
>>> maxima = []
>>> for i in range(0,1000) :
...    a = np.random.normal(mu, beta, 1000)
...    means.append(a.mean())
...    maxima.append(a.max())
>>> count, bins, ignored = plt.hist(maxima, 30, density=True)
>>> beta = np.std(maxima) * np.sqrt(6) / np.pi
>>> mu = np.mean(maxima) - 0.57721*beta
```

(continues on next page)



(continued from previous page)

```
>>> plt.plot(bins, (1/beta)*np.exp(-(bins - mu)/beta)
...          * np.exp(-np.exp(-(bins - mu)/beta)),
...          linewidth=2, color='r')
>>> plt.plot(bins, 1/(beta * np.sqrt(2 * np.pi))
...          * np.exp(-(bins - mu)**2 / (2 * beta**2)),
...          linewidth=2, color='g')
>>> plt.show()
```



method

`RandomState.`**`hypergeometric`**(*ngood*, *nbad*, *nsample*, *size=None*)

Draw samples from a Hypergeometric distribution.

Samples are drawn from a hypergeometric distribution with specified parameters, *ngood* (ways to make a good selection), *nbad* (ways to make a bad selection), and *nsample* (number of items sampled, which is less than or equal to the sum `ngood + nbad`).

---

**Note:** New code should use the `hypergeometric` method of a `default_rng()` instance instead; see *random-quick-start*.

---

**Parameters**

**ngood**

[int or array_like of ints] Number of ways to make a good selection. Must be nonnegative.

**nbad**

[int or array_like of ints] Number of ways to make a bad selection. Must be nonnegative.

**nsample**

[int or array_like of ints] Number of items sampled. Must be at least 1 and at most `ngood + nbad`.

**size**

[int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if *ngood*, *nbad*, and *nsample* are all scalars. Otherwise, `np.broadcast(ngood, nbad, nsample).size` samples are drawn.

**Returns**

**out**

[ndarray or scalar] Drawn samples from the parameterized hypergeometric distribution. Each sample is the number of good items within a randomly selected subset of size *nsample* taken from a set of *ngood* good items and *nbad* bad items.

**See also:**

**`scipy.stats.hypergeom`**

probability density function, distribution or cumulative density function, etc.

***`Generator.hypergeometric`***

which should be used for new code.

## Notes

The probability density for the Hypergeometric distribution is

$$P(x) = \frac{\binom{g}{x}\binom{b}{n-x}}{\binom{g+b}{n}},$$

where $0 \le x \le n$ and $n - b \le x \le g$

for P(x) the probability of `x` good results in the drawn sample, g = *ngood*, b = *nbad*, and n = *nsample*.

Consider an urn with black and white marbles in it, *ngood* of them are black and *nbad* are white. If you draw *nsample* balls without replacement, then the hypergeometric distribution describes the distribution of black balls in the drawn sample.

Note that this distribution is very similar to the binomial distribution, except that in this case, samples are drawn without replacement, whereas in the Binomial case samples are drawn with replacement (or the sample space is infinite). As the sample space becomes large, this distribution approaches the binomial.

### References

[1], [2], [3]

### Examples

Draw samples from the distribution:

```
>>> ngood, nbad, nsamp = 100, 2, 10
# number of good, number of bad, and number of samples
>>> s = np.random.hypergeometric(ngood, nbad, nsamp, 1000)
>>> from matplotlib.pyplot import hist
>>> hist(s)
#   note that it is very unlikely to grab both bad items
```

Suppose you have an urn with 15 white and 15 black marbles. If you pull 15 marbles at random, how likely is it that 12 or more of them are one color?

```
>>> s = np.random.hypergeometric(15, 15, 15, 100000)
>>> sum(s>=12)/100000. + sum(s<=3)/100000.
#   answer = 0.003 ... pretty unlikely!
```

method

`RandomState.`**`laplace`**(*loc=0.0*, *scale=1.0*, *size=None*)

Draw samples from the Laplace or double exponential distribution with specified location (or mean) and scale (decay).

The Laplace distribution is similar to the Gaussian/normal distribution, but is sharper at the peak and has fatter tails. It represents the difference between two independent, identically distributed exponential random variables.

---

**Note:** New code should use the `laplace` method of a `default_rng()` instance instead; see *random-quick-start*.

---

**Parameters**

**loc**

[float or array_like of floats, optional] The position, $\mu$, of the distribution peak. Default is 0.

**scale**

[float or array_like of floats, optional] $\lambda$, the exponential decay. Default is 1. Must be non-negative.

**size**

[int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `loc` and `scale` are both scalars. Otherwise, `np.broadcast(loc, scale).size` samples are drawn.

**Returns**

**out**

[ndarray or scalar] Drawn samples from the parameterized Laplace distribution.

**See also:**

*`Generator.laplace`*

which should be used for new code.

### Notes

It has the probability density function

$$f(x;\mu,\lambda) = \frac{1}{2\lambda}\exp\left(-\frac{|x-\mu|}{\lambda}\right).$$

The first law of Laplace, from 1774, states that the frequency of an error can be expressed as an exponential function of the absolute magnitude of the error, which leads to the Laplace distribution. For many problems in economics and health sciences, this distribution seems to model the data better than the standard Gaussian distribution.

### References

[1], [2], [3], [4]

### Examples

Draw samples from the distribution

```
>>> loc, scale = 0., 1.
>>> s = np.random.laplace(loc, scale, 1000)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 30, density=True)
>>> x = np.arange(-8., 8., .01)
>>> pdf = np.exp(-abs(x-loc)/scale)/(2.*scale)
>>> plt.plot(x, pdf)
```

Plot Gaussian for comparison:

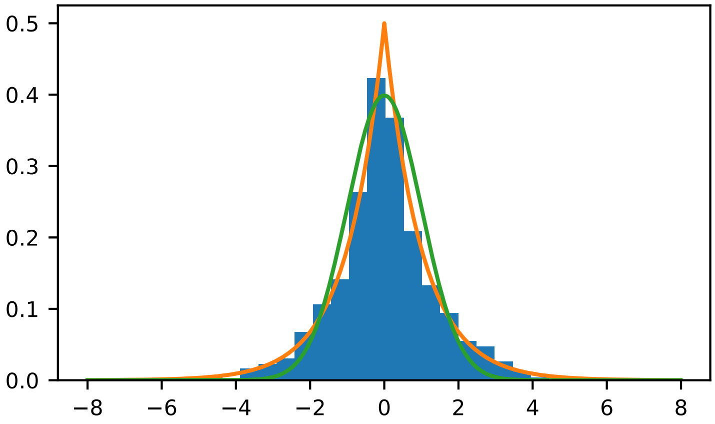
```
>>> g = (1/(scale * np.sqrt(2 * np.pi)) *
...      np.exp(-(x - loc)**2 / (2 * scale**2)))
>>> plt.plot(x,g)
```

method

`RandomState.`**`logistic`**(*loc=0.0*, *scale=1.0*, *size=None*)

Draw samples from a logistic distribution.

Samples are drawn from a logistic distribution with specified parameters, loc (location or mean, also median), and scale (>0).



---

**Note:** New code should use the `logistic` method of a `default_rng()` instance instead; see *random-quick-start*.

---

**Parameters**

**loc**

[float or array_like of floats, optional] Parameter of the distribution. Default is 0.

**scale**

[float or array_like of floats, optional] Parameter of the distribution. Must be non-negative. Default is 1.

**size**

[int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `loc` and `scale` are both scalars. Otherwise, `np.broadcast(loc, scale).size` samples are drawn.

**Returns**

**out**

[ndarray or scalar] Drawn samples from the parameterized logistic distribution.

**See also:**

`scipy.stats.logistic`

probability density function, distribution or cumulative density function, etc.

*`Generator.logistic`*

which should be used for new code.

### Notes

The probability density for the Logistic distribution is

$$P(x) = P(x) = \frac{e^{-(x-\mu)/s}}{s(1+e^{-(x-\mu)/s})^2},$$

where $\mu$ = location and $s$ = scale.

The Logistic distribution is used in Extreme Value problems where it can act as a mixture of Gumbel distributions, in Epidemiology, and by the World Chess Federation (FIDE) where it is used in the Elo ranking system, assuming the performance of each player is a logistically distributed random variable.

### References

[1], [2], [3]

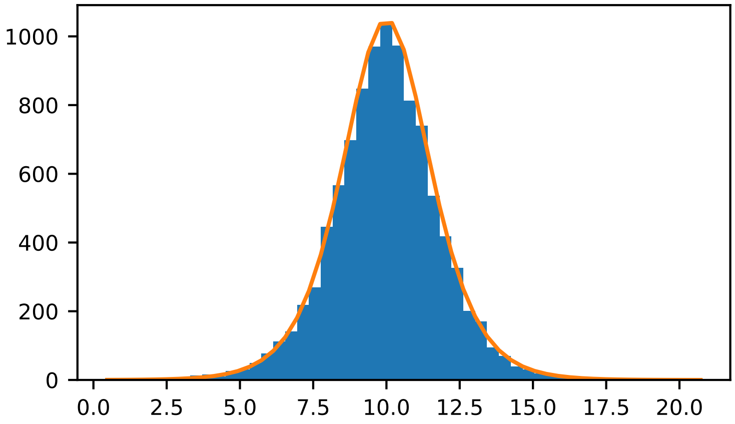
### Examples

Draw samples from the distribution:

```
>>> loc, scale = 10, 1
>>> s = np.random.logistic(loc, scale, 10000)
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, bins=50)
```

# plot against distribution

```
>>> def logist(x, loc, scale):
...     return np.exp((loc-x)/scale)/(scale*(1+np.exp((loc-x)/scale))**2)
>>> lgst_val = logist(bins, loc, scale)
>>> plt.plot(bins, lgst_val * count.max() / lgst_val.max())
>>> plt.show()
```



method

`RandomState.`**`lognormal`**`(`*mean=0.0*, *sigma=1.0*, *size=None*`)`

Draw samples from a log-normal distribution.

Draw samples from a log-normal distribution with specified mean, standard deviation, and array shape. Note that the mean and standard deviation are not the values for the distribution itself, but of the underlying normal distribution it is derived from.

---

**Note:** New code should use the `lognormal` method of a `default_rng()` instance instead; see *random-quick-start*.

---

**Parameters**

**mean**

[float or array_like of floats, optional] Mean value of the underlying normal distribution. Default is 0.

**sigma**

[float or array_like of floats, optional] Standard deviation of the underlying normal distribution. Must be non-negative. Default is 1.

**size**

[int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `mean` and `sigma` are both scalars. Otherwise, `np.broadcast(mean, sigma).size` samples are drawn.

**Returns**

**out**

[ndarray or scalar] Drawn samples from the parameterized log-normal distribution.

**See also:**

**`scipy.stats.lognorm`**

probability density function, distribution, cumulative density function, etc.

***`Generator.lognormal`***

which should be used for new code.

## Notes

A variable *x* has a log-normal distribution if *log(x)* is normally distributed. The probability density function for the log-normal distribution is:

$$p(x) = \frac{1}{\sigma x \sqrt{2\pi}} e^{(-\frac{(ln(x)-\mu)^2}{2\sigma^2})}$$

where $\mu$ is the mean and $\sigma$ is the standard deviation of the normally distributed logarithm of the variable. A log-normal distribution results if a random variable is the *product* of a large number of independent, identically-distributed variables in the same way that a normal distribution results if the variable is the *sum* of a large number of independent, identically-distributed variables.

### References

[1], [2]

### Examples

Draw samples from the distribution:

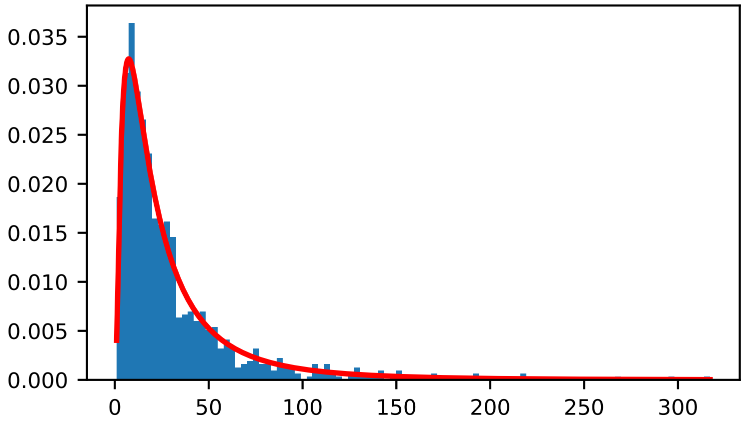
```
>>> mu, sigma = 3., 1. # mean and standard deviation
>>> s = np.random.lognormal(mu, sigma, 1000)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 100, density=True, align='mid')
```

```
>>> x = np.linspace(min(bins), max(bins), 10000)
>>> pdf = (np.exp(-(np.log(x) - mu)**2 / (2 * sigma**2))
...        / (x * sigma * np.sqrt(2 * np.pi)))
```

```
>>> plt.plot(x, pdf, linewidth=2, color='r')
>>> plt.axis('tight')
>>> plt.show()
```



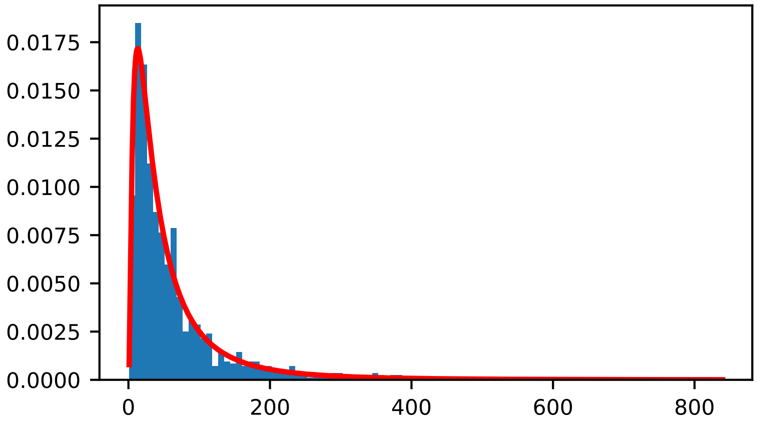
Demonstrate that taking the products of random samples from a uniform distribution can be fit well by a log-normal probability density function.

```
>>> # Generate a thousand samples: each is the product of 100 random
>>> # values, drawn from a normal distribution.
>>> b = []
>>> for i in range(1000):
...    a = 10. + np.random.standard_normal(100)
...    b.append(np.product(a))
```

```
>>> b = np.array(b) / np.min(b) # scale values to be positive
>>> count, bins, ignored = plt.hist(b, 100, density=True, align='mid')
>>> sigma = np.std(np.log(b))
>>> mu = np.mean(np.log(b))
```

```
>>> x = np.linspace(min(bins), max(bins), 10000)
>>> pdf = (np.exp(-(np.log(x) - mu)**2 / (2 * sigma**2))
...        / (x * sigma * np.sqrt(2 * np.pi)))
```

```
>>> plt.plot(x, pdf, color='r', linewidth=2)
>>> plt.show()
```



method

`RandomState.`**`logseries`**(*p*, *size=None*)

Draw samples from a logarithmic series distribution.

Samples are drawn from a log series distribution with specified shape parameter, 0 < `p` < 1.

**Note:** New code should use the `logseries` method of a `default_rng()` instance instead; see *random-quick-start*.

**Parameters**

**p**

[float or array_like of floats] Shape parameter for the distribution. Must be in the range (0, 1).

**size**

[int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `p` is a scalar. Otherwise, `np.array(p).size` samples are drawn.

**Returns**

**out**

[ndarray or scalar] Drawn samples from the parameterized logarithmic series distribution.

**See also:**

`scipy.stats.logser`

probability density function, distribution or cumulative density function, etc.

*`Generator.logseries`*

which should be used for new code.

### Notes

The probability density for the Log Series distribution is

$$P(k) = \frac{-p^k}{k \ln(1-p)},$$

where p = probability.

The log series distribution is frequently used to represent species richness and occurrence, first proposed by Fisher, Corbet, and Williams in 1943 [2]. It may also be used to model the numbers of occupants seen in cars [3].

### References

[1], [2], [3], [4]

### Examples

Draw samples from the distribution:

```
>>> a = .6
>>> s = np.random.logseries(a, 10000)
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s)
```

# plot against distribution

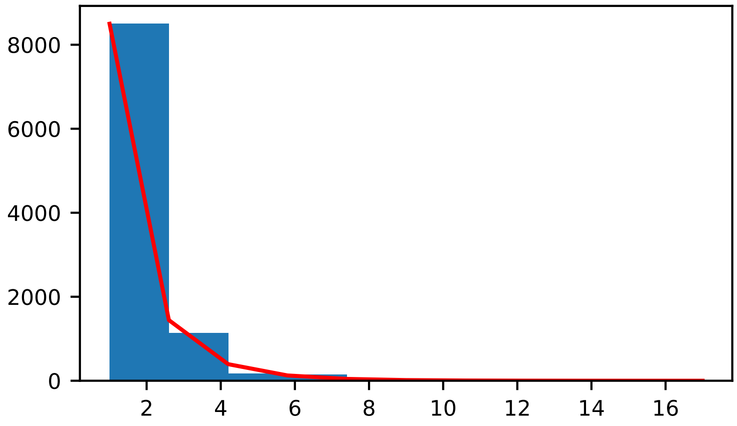
```
>>> def logseries(k, p):
...     return -p**k/(k*np.log(1-p))
>>> plt.plot(bins, logseries(bins, a)*count.max()/
...          logseries(bins, a).max(), 'r')
>>> plt.show()
```

method

`RandomState`.**`multinomial`**(*n*, *pvals*, *size=None*)

Draw samples from a multinomial distribution.

The multinomial distribution is a multivariate generalization of the binomial distribution. Take an experiment with one of `p` possible outcomes. An example of such an experiment is throwing a dice, where the outcome can be 1 through 6. Each sample drawn from the distribution represents *n* such experiments. Its values, `X_i = [X_0, X_1, ..., X_p]`, represent the number of times the outcome was `i`.



---

**Note:** New code should use the `multinomial` method of a `default_rng()` instance instead; see *random-quick-start*.

---

**Parameters**

**n**

[int] Number of experiments.

**pvals**

[sequence of floats, length p] Probabilities of each of the `p` different outcomes. These must sum to 1 (however, the last element is always assumed to account for the remaining probability, as long as `sum(pvals[:-1]) <= 1)`.

**size**

[int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. Default is None, in which case a single value is returned.

**Returns**

**out**

[ndarray] The drawn samples, of shape *size*, if that was provided. If not, the shape is `(N,)`.

In other words, each entry `out[i,j,...,:]` is an N-dimensional value drawn from the distribution.

**See also:**

`Generator.multinomial`

which should be used for new code.

### Examples

Throw a dice 20 times:

```
>>> np.random.multinomial(20, [1/6.]*6, size=1)
array([[4, 1, 7, 5, 2, 1]]) # random
```

It landed 4 times on 1, once on 2, etc.

Now, throw the dice 20 times, and 20 times again:

```
>>> np.random.multinomial(20, [1/6.]*6, size=2)
array([[3, 4, 3, 3, 4, 3], # random
       [2, 4, 3, 4, 0, 7]])
```

For the first run, we threw 3 times 1, 4 times 2, etc. For the second, we threw 2 times 1, 4 times 2, etc.

A loaded die is more likely to land on number 6:

```
>>> np.random.multinomial(100, [1/7.]*5 + [2/7.])
array([11, 16, 14, 17, 16, 26]) # random
```

The probability inputs should be normalized. As an implementation detail, the value of the last entry is ignored and assumed to take up any leftover probability mass, but this should not be relied on. A biased coin which has twice as much weight on one side as on the other should be sampled like so:

```
>>> np.random.multinomial(100, [1.0 / 3, 2.0 / 3])  # RIGHT
array([38, 62]) # random
```

not like:

```
>>> np.random.multinomial(100, [1.0, 2.0])  # WRONG
Traceback (most recent call last):
ValueError: pvals < 0, pvals > 1 or pvals contains NaNs
```

method

`RandomState.`**`multivariate_normal`**(*mean*, *cov*, *size=None*, *check_valid='warn'*, *tol=1e-8*)

Draw random samples from a multivariate normal distribution.

The multivariate normal, multinormal or Gaussian distribution is a generalization of the one-dimensional normal distribution to higher dimensions. Such a distribution is specified by its mean and covariance matrix. These parameters are analogous to the mean (average or "center") and variance (standard deviation, or "width," squared) of the one-dimensional normal distribution.

**Note:** New code should use the `multivariate_normal` method of a `default_rng()` instance instead; see *random-quick-start*.

**Parameters**

**mean**

[1-D array_like, of length N] Mean of the N-dimensional distribution.

**cov**

[2-D array_like, of shape (N, N)] Covariance matrix of the distribution. It must be symmetric and positive-semidefinite for proper sampling.

**size**

[int or tuple of ints, optional] Given a shape of, for example, `(m,n,k)`, `m*n*k` samples are generated, and packed in an *m*-by-*n*-by-*k* arrangement. Because each sample is *N*-dimensional, the output shape is `(m,n,k,N)`. If no shape is specified, a single (*N*-D) sample is returned.

**check_valid**

[{ 'warn', 'raise', 'ignore' }, optional] Behavior when the covariance matrix is not positive semidefinite.

**tol**

[float, optional] Tolerance when checking the singular values in covariance matrix. cov is cast to double before the check.

**Returns**

**out**

[ndarray] The drawn samples, of shape *size*, if that was provided. If not, the shape is `(N,)`.

In other words, each entry `out[i,j,...,:]` is an N-dimensional value drawn from the distribution.

**See also:**

***`Generator.multivariate_normal`***

which should be used for new code.

## Notes

The mean is a coordinate in N-dimensional space, which represents the location where samples are most likely to be generated. This is analogous to the peak of the bell curve for the one-dimensional or univariate normal distribution.

Covariance indicates the level to which two variables vary together. From the multivariate normal distribution, we draw N-dimensional samples, $X = [x_1, x_2, ...x_N]$. The covariance matrix element $C_{ij}$ is the covariance of $x_i$ and $x_j$. The element $C_{ii}$ is the variance of $x_i$ (i.e. its "spread").

Instead of specifying the full covariance matrix, popular approximations include:

- Spherical covariance (*cov* is a multiple of the identity matrix)
- Diagonal covariance (*cov* has non-negative elements, and only on the diagonal)

This geometrical property can be seen in two dimensions by plotting generated data-points:

```
>>> mean = [0, 0]
>>> cov = [[1, 0], [0, 100]]  # diagonal covariance
```

Diagonal covariance means that points are oriented along x or y-axis:

```
>>> import matplotlib.pyplot as plt
>>> x, y = np.random.multivariate_normal(mean, cov, 5000).T
>>> plt.plot(x, y, 'x')
>>> plt.axis('equal')
>>> plt.show()
```

Note that the covariance matrix must be positive semidefinite (a.k.a. nonnegative-definite). Otherwise, the behavior of this method is undefined and backwards compatibility is not guaranteed.

### References

[1], [2]

### Examples

```
>>> mean = (1, 2)
>>> cov = [[1, 0], [0, 1]]
>>> x = np.random.multivariate_normal(mean, cov, (3, 3))
>>> x.shape
(3, 3, 2)
```

The following is probably true, given that 0.6 is roughly twice the standard deviation:

```
>>> list((x[0,0,:] - mean) < 0.6)
[True, True] # random
```

method

`RandomState.`**`negative_binomial`**(*n*, *p*, *size=None*)

Draw samples from a negative binomial distribution.

Samples are drawn from a negative binomial distribution with specified parameters, *n* successes and *p* probability of success where *n* is > 0 and *p* is in the interval [0, 1].

---

**Note:** New code should use the `negative_binomial` method of a `default_rng()` instance instead; see *random-quick-start*.

---

**Parameters**

**n**

[float or array_like of floats] Parameter of the distribution, > 0.

**p**

[float or array_like of floats] Parameter of the distribution, >= 0 and <=1.

**size**

[int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `n` and `p` are both scalars. Otherwise, `np.broadcast(n, p).size` samples are drawn.

**Returns**

**out**

[ndarray or scalar] Drawn samples from the parameterized negative binomial distribution, where each sample is equal to N, the number of failures that occurred before a total of n successes was reached.

**See also:**

*`Generator.negative_binomial`*

which should be used for new code.

## Notes

The probability mass function of the negative binomial distribution is

$$P(N; n, p) = \frac{\Gamma(N + n)}{N!\Gamma(n)} p^n (1 - p)^N,$$

where $n$ is the number of successes, $p$ is the probability of success, $N + n$ is the number of trials, and $\Gamma$ is the gamma function. When $n$ is an integer, $\frac{\Gamma(N+n)}{N!\Gamma(n)} = \binom{N+n-1}{N}$, which is the more common form of this term in the the pmf. The negative binomial distribution gives the probability of N failures given n successes, with a success on the last trial.

If one throws a die repeatedly until the third time a "1" appears, then the probability distribution of the number of non-"1"s that appear before the third "1" is a negative binomial distribution.

## References

[1], [2]

## Examples

Draw samples from the distribution:

A real world example. A company drills wild-cat oil exploration wells, each with an estimated probability of success of 0.1. What is the probability of having one success for each successive well, that is what is the probability of a single success after drilling 5 wells, after 6 wells, etc.?

```
>>> s = np.random.negative_binomial(1, 0.1, 100000)
>>> for i in range(1, 11):
...     probability = sum(s<i) / 100000.
...     print(i, "wells drilled, probability of one success =", probability)
```

method

`RandomState.`**`noncentral_chisquare`**(*df*, *nonc*, *size=None*)

Draw samples from a noncentral chi-square distribution.

The noncentral $\chi^2$ distribution is a generalization of the $\chi^2$ distribution.

---

**Note:** New code should use the `noncentral_chisquare` method of a `default_rng()` instance instead; see *random-quick-start*.

---

**Parameters**

**df**

[float or array_like of floats] Degrees of freedom, must be > 0.

Changed in version 1.10.0: Earlier NumPy versions required dfnum > 1.

**nonc**

[float or array_like of floats] Non-centrality, must be non-negative.

**size**

[int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `df` and `nonc` are both scalars. Otherwise, `np.broadcast(df, nonc).size` samples are drawn.

**Returns**

**out**

[ndarray or scalar] Drawn samples from the parameterized noncentral chi-square distribution.

**See also:**

*`Generator.noncentral_chisquare`*

which should be used for new code.

## Notes

The probability density function for the noncentral Chi-square distribution is

$$P(x;df,nonc) = \sum_{i=0}^{\infty} \frac{e^{-nonc/2}(nonc/2)^{i}}{i!} P_{Y_{df+2i}}(x),$$

where $Y_q$ is the Chi-square with q degrees of freedom.

## References

[1]

## Examples

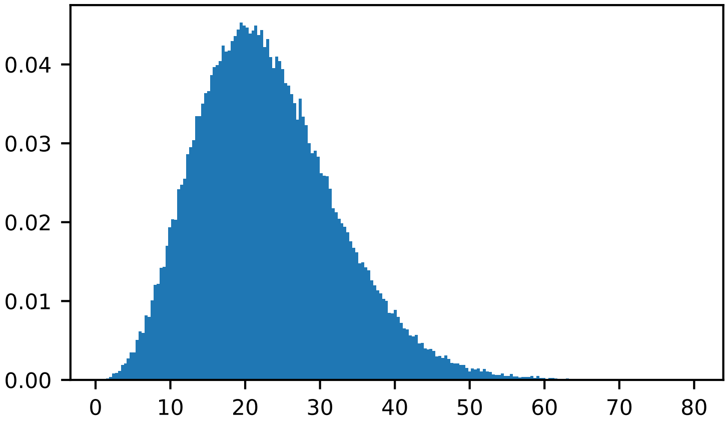
Draw values from the distribution and plot the histogram

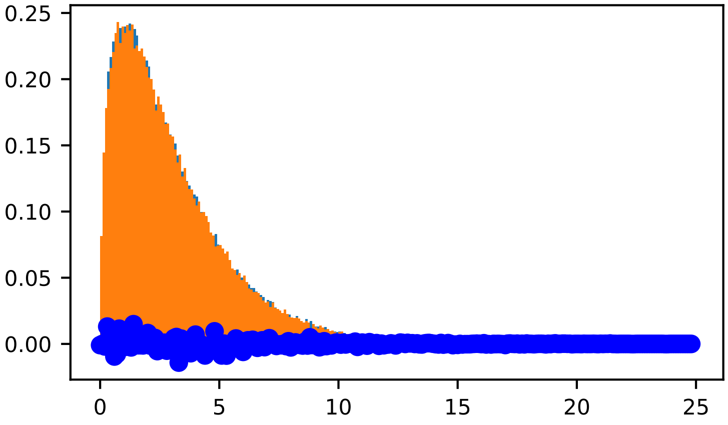
```
>>> import matplotlib.pyplot as plt
>>> values = plt.hist(np.random.noncentral_chisquare(3, 20, 100000),
...                   bins=200, density=True)
>>> plt.show()
```

Draw values from a noncentral chisquare with very small noncentrality, and compare to a chisquare.

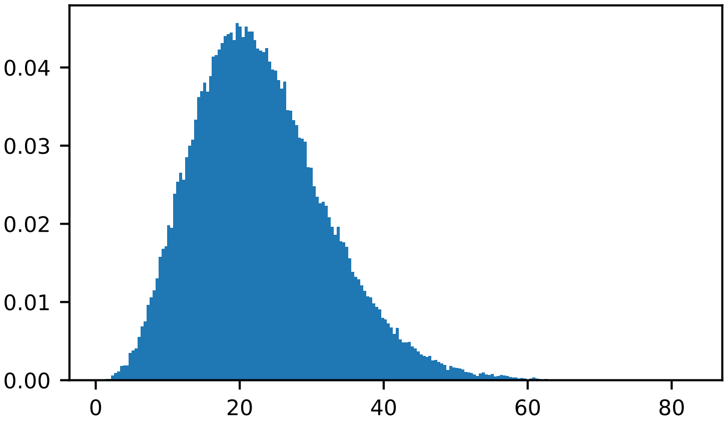
```
>>> plt.figure()
>>> values = plt.hist(np.random.noncentral_chisquare(3, .0000001, 100000),
...                   bins=np.arange(0., 25, .1), density=True)
>>> values2 = plt.hist(np.random.chisquare(3, 100000),
...                    bins=np.arange(0., 25, .1), density=True)
>>> plt.plot(values[1][0:-1], values[0]-values2[0], 'ob')
>>> plt.show()
```

Demonstrate how large values of non-centrality lead to a more symmetric distribution.


0.04
0.03
0.02
0.01
0.00
0
10
20
30
40
50
60
70
80


0.25
0.20
0.15
0.10
0.05
0.00
0
5
10
15
20
25

```
>>> plt.figure()
>>> values = plt.hist(np.random.noncentral_chisquare(3, 20, 100000),
...                   bins=200, density=True)
>>> plt.show()
```



method

`RandomState.`**`noncentral_f`**`(`*dfnum*, *dfden*, *nonc*, *size=None*`)`

Draw samples from the noncentral F distribution.

Samples are drawn from an F distribution with specified parameters, *dfnum* (degrees of freedom in numerator) and *dfden* (degrees of freedom in denominator), where both parameters > 1. *nonc* is the non-centrality parameter.

---

**Note:** New code should use the `noncentral_f` method of a `default_rng()` instance instead; see *random-quick-start*.

---

**Parameters**

**dfnum**

[float or array_like of floats] Numerator degrees of freedom, must be > 0.

Changed in version 1.14.0: Earlier NumPy versions required dfnum > 1.

**dfden**

[float or array_like of floats] Denominator degrees of freedom, must be > 0.

**nonc**

[float or array_like of floats] Non-centrality parameter, the sum of the squares of the numerator means, must be >= 0.

**size**

[int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `dfnum`, `dfden`, and `nonc` are all scalars. Otherwise, `np.broadcast(dfnum, dfden, nonc).size` samples are drawn.

**Returns**

**out**

[ndarray or scalar] Drawn samples from the parameterized noncentral Fisher distribution.

**See also:**

***Generator.noncentral_f***

which should be used for new code.

### Notes

When calculating the power of an experiment (power = probability of rejecting the null hypothesis when a specific alternative is true) the non-central F statistic becomes important. When the null hypothesis is true, the F statistic follows a central F distribution. When the null hypothesis is not true, then it follows a non-central F statistic.

### References

[1], [2]

### Examples

In a study, testing for a specific alternative to the null hypothesis requires use of the Noncentral F distribution. We need to calculate the area in the tail of the distribution that exceeds the value of the F distribution for the null hypothesis. We'll plot the two probability distributions for comparison.

```
>>> dfnum = 3 # between group deg of freedom
>>> dfden = 20 # within groups degrees of freedom
>>> nonc = 3.0
>>> nc_vals = np.random.noncentral_f(dfnum, dfden, nonc, 1000000)
>>> NF = np.histogram(nc_vals, bins=50, density=True)
>>> c_vals = np.random.f(dfnum, dfden, 1000000)
>>> F = np.histogram(c_vals, bins=50, density=True)
>>> import matplotlib.pyplot as plt
>>> plt.plot(F[1][1:], F[0])
>>> plt.plot(NF[1][1:], NF[0])
>>> plt.show()
```

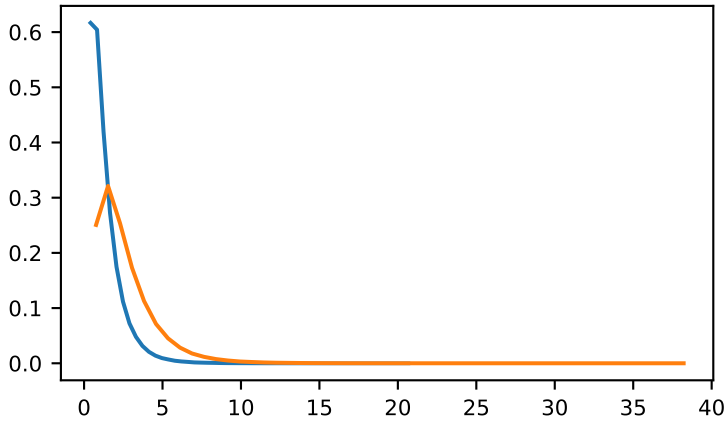
method

`RandomState.`**`normal`**(*loc=0.0*, *scale=1.0*, *size=None*)

Draw random samples from a normal (Gaussian) distribution.

The probability density function of the normal distribution, first derived by De Moivre and 200 years later by both Gauss and Laplace independently [2], is often called the bell curve because of its characteristic shape (see the example below).

The normal distributions occurs often in nature. For example, it describes the commonly occurring distribution of samples influenced by a large number of tiny, random disturbances, each with its own unique distribution [2].



---

**Note:** New code should use the `normal` method of a `default_rng()` instance instead; see *random-quick-start*.

---

**Parameters**

**loc**

[float or array_like of floats] Mean ("centre") of the distribution.

**scale**

[float or array_like of floats] Standard deviation (spread or "width") of the distribution. Must be non-negative.

**size**

[int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `loc` and `scale` are both scalars. Otherwise, `np.broadcast(loc, scale).size` samples are drawn.

**Returns**

**out**

[ndarray or scalar] Drawn samples from the parameterized normal distribution.

**See also:**

`scipy.stats.norm`

probability density function, distribution or cumulative density function, etc.

*`Generator.normal`*

which should be used for new code.

### Notes

The probability density for the Gaussian distribution is

$$p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}},$$

where $\mu$ is the mean and $\sigma$ the standard deviation. The square of the standard deviation, $\sigma^2$, is called the variance.

The function has its peak at the mean, and its "spread" increases with the standard deviation (the function reaches 0.607 times its maximum at $x + \sigma$ and $x - \sigma$ [2]). This implies that normal is more likely to return samples lying close to the mean, rather than those far away.

### References

[1], [2]

### Examples

Draw samples from the distribution:

```
>>> mu, sigma = 0, 0.1 # mean and standard deviation
>>> s = np.random.normal(mu, sigma, 1000)
```

Verify the mean and the variance:

```
>>> abs(mu - np.mean(s))
0.0  # may vary
```

```
>>> abs(sigma - np.std(s, ddof=1))
0.1  # may vary
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 30, density=True)
>>> plt.plot(bins, 1/(sigma * np.sqrt(2 * np.pi)) *
...                np.exp( - (bins - mu)**2 / (2 * sigma**2) ),
...          linewidth=2, color='r')
>>> plt.show()
```

Two-by-four array of samples from N(3, 6.25):

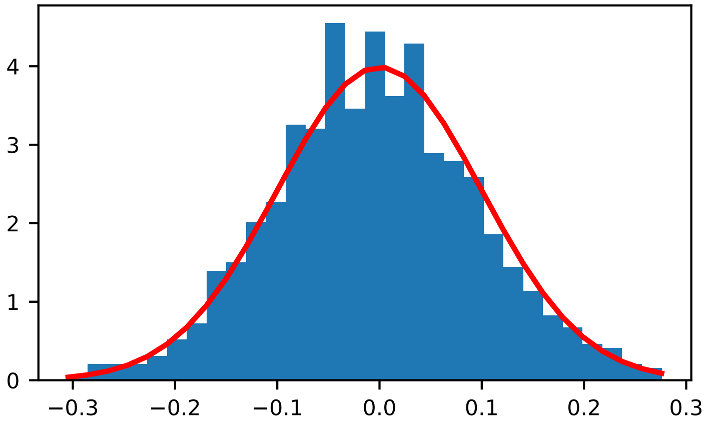
```
>>> np.random.normal(3, 2.5, size=(2, 4))
array([[-4.49401501,  4.00950034, -1.81814867,  7.29718677],   # random
       [ 0.39924804,  4.68456316,  4.99394529,  4.84057254]])  # random
```

method

`RandomState.`**`pareto`**(*a*, *size=None*)

Draw samples from a Pareto II or Lomax distribution with specified shape.

The Lomax or Pareto II distribution is a shifted Pareto distribution. The classical Pareto distribution can be obtained from the Lomax distribution by adding 1 and multiplying by the scale parameter `m` (see Notes). The smallest value of the Lomax distribution is zero while for the classical Pareto distribution it is `mu`, where the standard Pareto



distribution has location `mu = 1`. Lomax can also be considered as a simplified version of the Generalized Pareto distribution (available in SciPy), with the scale set to one and the location set to zero.

The Pareto distribution must be greater than zero, and is unbounded above. It is also known as the “80-20 rule”. In this distribution, 80 percent of the weights are in the lowest 20 percent of the range, while the other 20 percent fill the remaining 80 percent of the range.

---

**Note:** New code should use the `pareto` method of a `default_rng()` instance instead; see *random-quick-start*.

---

**Parameters**

**a**

[float or array_like of floats] Shape of the distribution. Must be positive.

**size**

[int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `a` is a scalar. Otherwise, `np.array(a).size` samples are drawn.

**Returns**

**out**

[ndarray or scalar] Drawn samples from the parameterized Pareto distribution.

**See also:**

**`scipy.stats.lomax`**

probability density function, distribution or cumulative density function, etc.

**`scipy.stats.genpareto`**

probability density function, distribution or cumulative density function, etc.

`Generator.pareto`

which should be used for new code.

### Notes

The probability density for the Pareto distribution is

$$p(x) = \frac{am^a}{x^{a+1}}$$

where $a$ is the shape and $m$ the scale.

The Pareto distribution, named after the Italian economist Vilfredo Pareto, is a power law probability distribution useful in many real world problems. Outside the field of economics it is generally referred to as the Bradford distribution. Pareto developed the distribution to describe the distribution of wealth in an economy. It has also found use in insurance, web page access statistics, oil field sizes, and many other problems, including the download frequency for projects in Sourceforge [1]. It is one of the so-called "fat-tailed" distributions.

### References

[1], [2], [3], [4]

### Examples

Draw samples from the distribution:

```
>>> a, m = 3., 2.  # shape and mode
>>> s = (np.random.pareto(a, 1000) + 1) * m
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, _ = plt.hist(s, 100, density=True)
>>> fit = a*m**a / bins**(a+1)
>>> plt.plot(bins, max(count)*fit/max(fit), linewidth=2, color='r')
>>> plt.show()
```

method

`RandomState.`**`poisson`**(*lam=1.0*, *size=None*)

Draw samples from a Poisson distribution.

The Poisson distribution is the limit of the binomial distribution for large N.

---

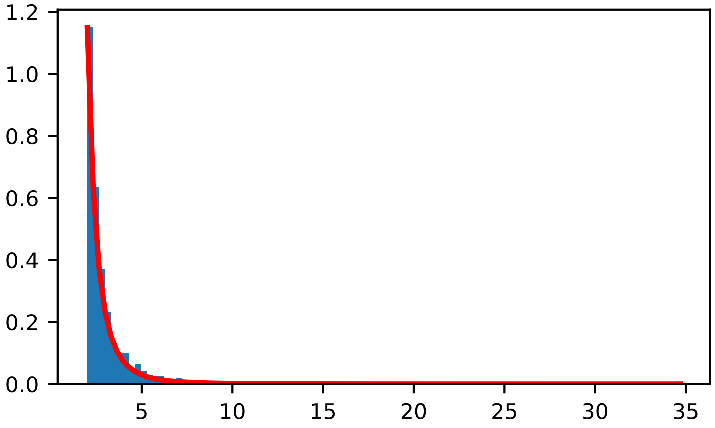
**Note:** New code should use the `poisson` method of a `default_rng()` instance instead; see *random-quick-start*.

---

#### Parameters

**lam**

[float or array_like of floats] Expectation of interval, must be >= 0. A sequence of expectation intervals must be broadcastable over the requested size.



**size**

[int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `lam` is a scalar. Otherwise, `np.array(lam).size` samples are drawn.

**Returns**

**out**

[ndarray or scalar] Drawn samples from the parameterized Poisson distribution.

**See also:**

**`Generator.poisson`**

which should be used for new code.

## Notes

The Poisson distribution

$$f(k; \lambda) = \frac{\lambda^k e^{-\lambda}}{k!}$$

For events with an expected separation $\lambda$ the Poisson distribution $f(k; \lambda)$ describes the probability of $k$ events occurring within the observed interval $\lambda$.

Because the output is limited to the range of the C int64 type, a ValueError is raised when *lam* is within 10 sigma of the maximum representable value.

### References

[1], [2]

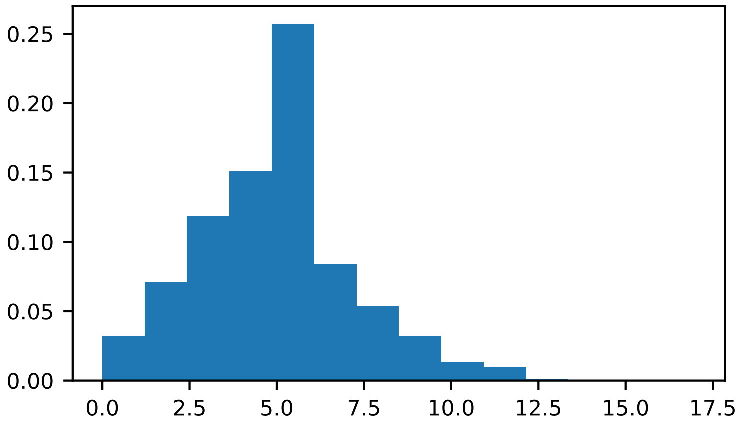
### Examples

Draw samples from the distribution:

```
>>> import numpy as np
>>> s = np.random.poisson(5, 10000)
```

Display histogram of the sample:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 14, density=True)
>>> plt.show()
```



Draw each 100 values for lambda 100 and 500:

```
>>> s = np.random.poisson(lam=(100., 500.), size=(100, 2))
```

method

`RandomState.`**`power`**`(`*a*, *size=None*`)`

Draws samples in [0, 1] from a power distribution with positive exponent a - 1.

Also known as the power function distribution.

**Note:** New code should use the `power` method of a `default_rng()` instance instead; see *random-quick-start*.

**Parameters**

**a**

[float or array_like of floats] Parameter of the distribution. Must be non-negative.

**size**

[int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `a` is a scalar. Otherwise, `np.array(a).size` samples are drawn.

**Returns**

**out**

[ndarray or scalar] Drawn samples from the parameterized power distribution.

**Raises**

**ValueError**

If a < 1.

**See also:**

*`Generator.power`*

which should be used for new code.

## Notes

The probability density function is

$$P(x; a) = ax^{a-1}, 0 \le x \le 1, a>0.$$

The power function distribution is just the inverse of the Pareto distribution. It may also be seen as a special case of the Beta distribution.

It is used, for example, in modeling the over-reporting of insurance claims.

## References

[1], [2]

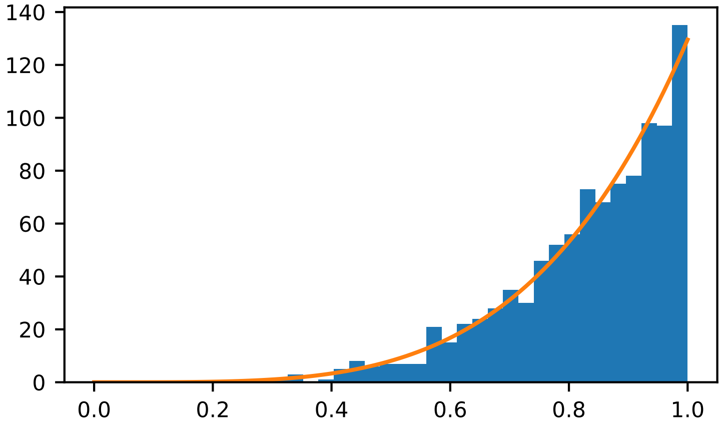
## Examples

Draw samples from the distribution:

```
>>> a = 5. # shape
>>> samples = 1000
>>> s = np.random.power(a, samples)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, bins=30)
>>> x = np.linspace(0, 1, 100)
>>> y = a*x**(a-1.)
>>> normed_y = samples*np.diff(bins)[0]*y
>>> plt.plot(x, normed_y)
>>> plt.show()
```



Compare the power function distribution to the inverse of the Pareto.

```
>>> from scipy import stats
>>> rvs = np.random.power(5, 1000000)
>>> rvsp = np.random.pareto(5, 1000000)
>>> xx = np.linspace(0,1,100)
>>> powpdf = stats.powerlaw.pdf(xx,5)
```

```
>>> plt.figure()
>>> plt.hist(rvs, bins=50, density=True)
>>> plt.plot(xx,powpdf,'r-')
>>> plt.title('np.random.power(5)')
```

```
>>> plt.figure()
>>> plt.hist(1./(1.+rvsp), bins=50, density=True)
>>> plt.plot(xx,powpdf,'r-')
>>> plt.title('inverse of 1 + np.random.pareto(5)')
```

```
>>> plt.figure()
>>> plt.hist(1./(1.+rvsp), bins=50, density=True)
>>> plt.plot(xx,powpdf,'r-')
>>> plt.title('inverse of stats.pareto(5)')
```

method

`RandomState.`**`rayleigh`**(*scale=1.0*, *size=None*)

Draw samples from a Rayleigh distribution.

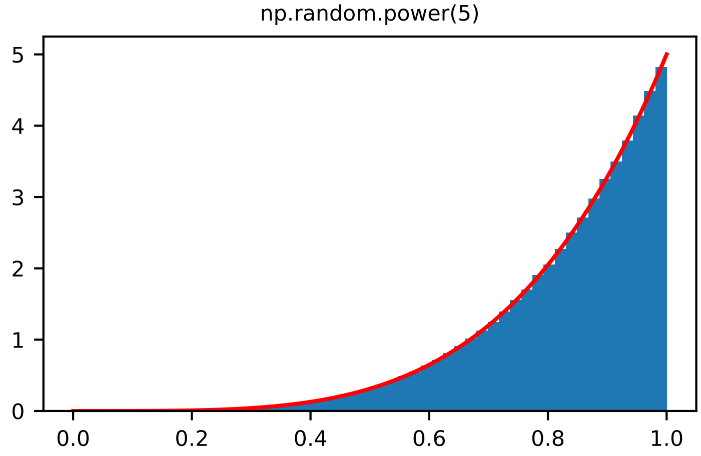
The $\chi$ and Weibull distributions are generalizations of the Rayleigh.

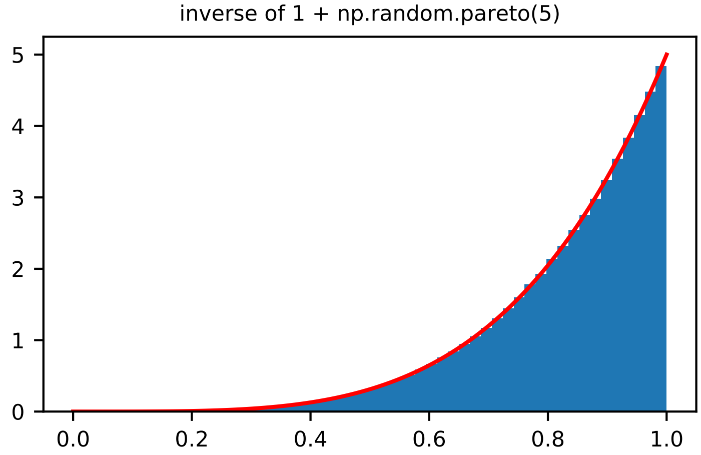
---

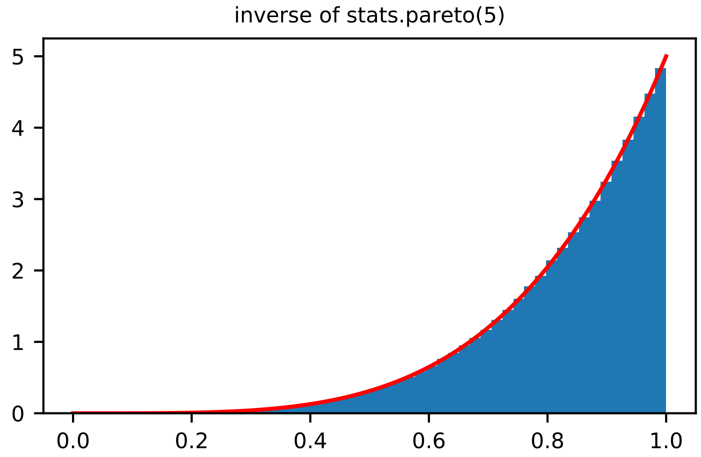
**Note:** New code should use the `rayleigh` method of a `default_rng()` instance instead; see *random-quick-start*.

---

**Parameters**


np.random.power(5)
0
1
2
3
4
5
0.0
0.2
0.4
0.6
0.8
1.0


inverse of 1 + np.random.pareto(5)
0
1
2
3
4
5
0.0
0.2
0.4
0.6
0.8
1.0



**scale**

[float or array_like of floats, optional] Scale, also equals the mode. Must be non-negative. Default is 1.

**size**

[int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `scale` is a scalar. Otherwise, `np.array(scale).size` samples are drawn.

**Returns**

**out**

[ndarray or scalar] Drawn samples from the parameterized Rayleigh distribution.

**See also:**

`Generator.rayleigh`

which should be used for new code.

## Notes

The probability density function for the Rayleigh distribution is

$$P(x;scale) = \frac{x}{scale^2}e^{\frac{-x^2}{2 \cdot scale^2}}$$

The Rayleigh distribution would arise, for example, if the East and North components of the wind velocity had identical zero-mean Gaussian distributions. Then the wind speed would have a Rayleigh distribution.

### References

[1], [2]

### Examples

Draw values from the distribution and plot the histogram

```
>>> from matplotlib.pyplot import hist
>>> values = hist(np.random.rayleigh(3, 100000), bins=200, density=True)
```

Wave heights tend to follow a Rayleigh distribution. If the mean wave height is 1 meter, what fraction of waves are likely to be larger than 3 meters?

```
>>> meanvalue = 1
>>> modevalue = np.sqrt(2 / np.pi) * meanvalue
>>> s = np.random.rayleigh(modevalue, 1000000)
```

The percentage of waves larger than 3 meters is:

```
>>> 100.*sum(s>3)/1000000.
0.087300000000000003 # random
```

method

`RandomState.`**`standard_cauchy`**`(`*size=None*`)`

Draw samples from a standard Cauchy distribution with mode = 0.

Also known as the Lorentz distribution.

---

**Note:** New code should use the `standard_cauchy` method of a `default_rng()` instance instead; see *random-quick-start*.

---

**Parameters**

**size**

[int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. Default is None, in which case a single value is returned.

**Returns**

**samples**

[ndarray or scalar] The drawn samples.

**See also:**

*`Generator.standard_cauchy`*

which should be used for new code.

### Notes

The probability density function for the full Cauchy distribution is

$$P(x; x_0, \gamma) = \frac{1}{\pi \gamma \bigl[ 1+(\frac{x-x_0}{\gamma})^2 \bigr] }$$

and the Standard Cauchy distribution just sets $x_0 = 0$ and $\gamma = 1$

The Cauchy distribution arises in the solution to the driven harmonic oscillator problem, and also describes spectral line broadening. It also describes the distribution of values at which a line tilted at a random angle will cut the x axis.

When studying hypothesis tests that assume normality, seeing how the tests perform on data from a Cauchy distribution is a good indicator of their sensitivity to a heavy-tailed distribution, since the Cauchy looks very much like a Gaussian distribution, but with heavier tails.

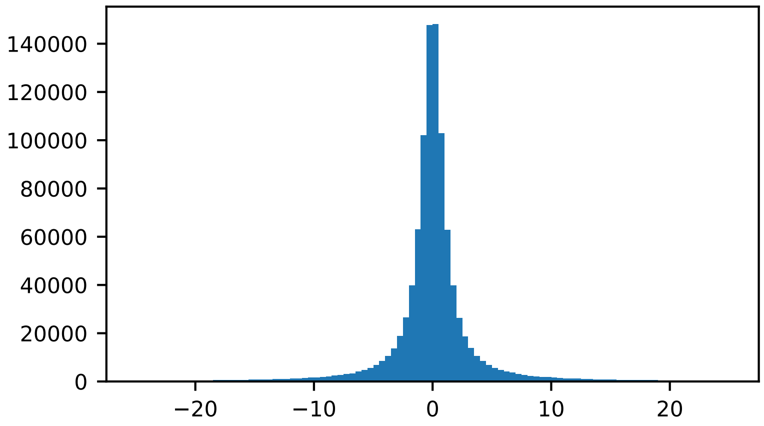
### References

[1], [2], [3]

### Examples

Draw samples and plot the distribution:

```
>>> import matplotlib.pyplot as plt
>>> s = np.random.standard_cauchy(1000000)
>>> s = s[(s>-25) & (s<25)]  # truncate distribution so it plots well
>>> plt.hist(s, bins=100)
>>> plt.show()
```



method

`RandomState.`**`standard_exponential`**(*size=None*)

Draw samples from the standard exponential distribution.

*standard_exponential* is identical to the exponential distribution with a scale parameter of 1.

---

**Note:** New code should use the `standard_exponential` method of a `default_rng()` instance instead; see *random-quick-start*.

---

**Parameters**

**size**

[int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. Default is None, in which case a single value is returned.

**Returns**

**out**

[float or ndarray] Drawn samples.

**See also:**

***Generator.standard_exponential***

which should be used for new code.

### Examples

Output a 3x8000 array:

```
>>> n = np.random.standard_exponential((3, 8000))
```

method

`RandomState.`**`standard_gamma`**(*shape*, *size=None*)

Draw samples from a standard Gamma distribution.

Samples are drawn from a Gamma distribution with specified parameters, shape (sometimes designated “k”) and scale=1.

---

**Note:** New code should use the `standard_gamma` method of a `default_rng()` instance instead; see *random-quick-start*.

---

**Parameters**

**shape**

[float or array_like of floats] Parameter, must be non-negative.

**size**

[int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `shape` is a scalar. Otherwise, `np.array(shape).size` samples are drawn.

**Returns**

**out**

[ndarray or scalar] Drawn samples from the parameterized standard gamma distribution.

**See also:**

`scipy.stats.gamma`

probability density function, distribution or cumulative density function, etc.

*`Generator.standard_gamma`*

which should be used for new code.

## Notes

The probability density for the Gamma distribution is

$$p(x) = x^{k-1}\frac{e^{-x/\theta}}{\theta^k\Gamma(k)},$$

where $k$ is the shape and $\theta$ the scale, and $\Gamma$ is the Gamma function.

The Gamma distribution is often used to model the times to failure of electronic components, and arises naturally in processes for which the waiting times between Poisson distributed events are relevant.

## References

[1], [2]

## Examples

Draw samples from the distribution:

```
>>> shape, scale = 2., 1. # mean and width
>>> s = np.random.standard_gamma(shape, 1000000)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> import scipy.special as sps
>>> count, bins, ignored = plt.hist(s, 50, density=True)
>>> y = bins**(shape-1) * ((np.exp(-bins/scale))/
...                       (sps.gamma(shape) * scale**shape))
>>> plt.plot(bins, y, linewidth=2, color='r')
>>> plt.show()
```

method

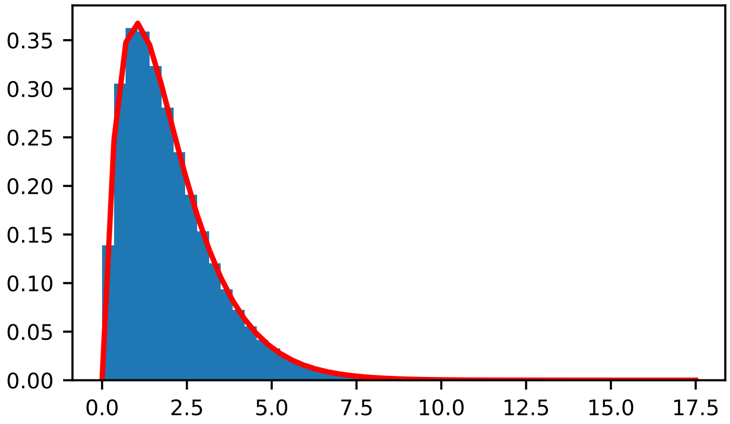
`RandomState.`**`standard_normal`**`(`*size=None*`)`

Draw samples from a standard Normal distribution (mean=0, stdev=1).

---

**Note:** New code should use the `standard_normal` method of a `default_rng()` instance instead; see *random-quick-start*.

---



**Parameters**

**size**

[int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. Default is None, in which case a single value is returned.

**Returns**

**out**

[float or ndarray] A floating-point array of shape `size` of drawn samples, or a single sample if `size` was not specified.

**See also:**

***normal***

Equivalent function with additional `loc` and `scale` arguments for setting the mean and standard deviation.

***Generator.standard_normal***

which should be used for new code.

## Notes

For random samples from $N(\mu, \sigma^2)$, use one of:

```
mu + sigma * np.random.standard_normal(size=...)
np.random.normal(mu, sigma, size=...)
```

### Examples

```
>>> np.random.standard_normal()
2.1923875335537315 #random
```

```
>>> s = np.random.standard_normal(8000)
>>> s
array([ 0.6888893 ,  0.78096262, -0.89086505, ...,  0.49876311,  # random
       -0.38672696, -0.4685006 ])                                # random
>>> s.shape
(8000,)
>>> s = np.random.standard_normal(size=(3, 4, 2))
>>> s.shape
(3, 4, 2)
```

Two-by-four array of samples from $N(3, 6.25)$:

```
>>> 3 + 2.5 * np.random.standard_normal(size=(2, 4))
array([[-4.49401501,  4.00950034, -1.81814867,  7.29718677],   # random
       [ 0.39924804,  4.68456316,  4.99394529,  4.84057254]])  # random
```

method

`RandomState.`**`standard_t`**(*df*, *size=None*)

Draw samples from a standard Student's t distribution with *df* degrees of freedom.

A special case of the hyperbolic distribution. As *df* gets large, the result resembles that of the standard normal distribution (*`standard_normal`*).

---

**Note:** New code should use the `standard_t` method of a `default_rng()` instance instead; see *random-quick-start*.

---

**Parameters**

**df**

[float or array_like of floats] Degrees of freedom, must be > 0.

**size**

[int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `df` is a scalar. Otherwise, `np.array(df).size` samples are drawn.

**Returns**

**out**

[ndarray or scalar] Drawn samples from the parameterized standard Student's t distribution.

**See also:**

***`Generator.standard_t`***

which should be used for new code.

### Notes

The probability density function for the t distribution is

$$P(x, df) = \frac{\Gamma(\frac{df+1}{2})}{\sqrt{\pi df}\Gamma(\frac{df}{2})}\Bigl( 1+\frac{x^2}{df} \Bigr)^{-(df+1)/2}$$

The t test is based on an assumption that the data come from a Normal distribution. The t test provides a way to test whether the sample mean (that is the mean calculated from the data) is a good estimate of the true mean.

The derivation of the t-distribution was first published in 1908 by William Gosset while working for the Guinness Brewery in Dublin. Due to proprietary issues, he had to publish under a pseudonym, and so he used the name Student.

### References

[1], [2]

### Examples

From Dalgaard page 83 [1], suppose the daily energy intake for 11 women in kilojoules (kJ) is:

```
>>> intake = np.array([5260., 5470, 5640, 6180, 6390, 6515, 6805, 7515, \
...                    7515, 8230, 8770])
```

Does their energy intake deviate systematically from the recommended value of 7725 kJ?

We have 10 degrees of freedom, so is the sample mean within 95% of the recommended value?

```
>>> s = np.random.standard_t(10, size=100000)
>>> np.mean(intake)
6753.636363636364
>>> intake.std(ddof=1)
1142.1232221373727
```

Calculate the t statistic, setting the ddof parameter to the unbiased value so the divisor in the standard deviation will be degrees of freedom, N-1.

```
>>> t = (np.mean(intake)-7725)/(intake.std(ddof=1)/np.sqrt(len(intake)))
>>> import matplotlib.pyplot as plt
>>> h = plt.hist(s, bins=100, density=True)
```

For a one-sided t-test, how far out in the distribution does the t statistic appear?

```
>>> np.sum(s<t) / float(len(s))
0.0090699999999999999  #random
```

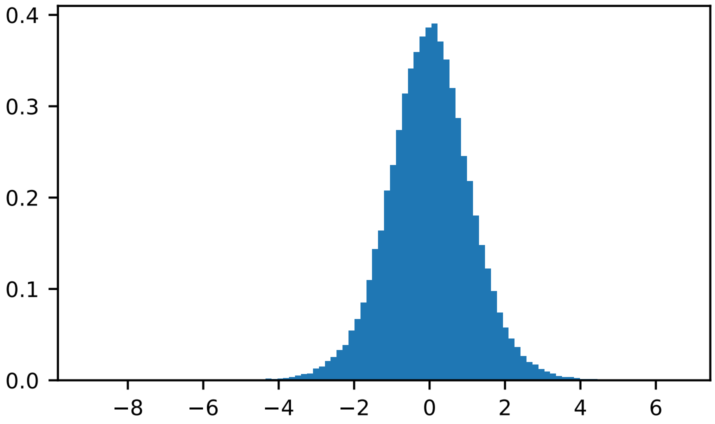
So the p-value is about 0.009, which says the null hypothesis has a probability of about 99% of being true.

method

`RandomState.`**`triangular`**(*left*, *mode*, *right*, *size=None*)

Draw samples from the triangular distribution over the interval `[left, right]`.

The triangular distribution is a continuous probability distribution with lower limit left, peak at mode, and upper limit right. Unlike the other distributions, these parameters directly define the shape of the pdf.



---

**Note:** New code should use the `triangular` method of a `default_rng()` instance instead; see *random-quick-start*.

---

**Parameters**

**left**

[float or array_like of floats] Lower limit.

**mode**

[float or array_like of floats] The value where the peak of the distribution occurs. The value must fulfill the condition `left <= mode <= right`.

**right**

[float or array_like of floats] Upper limit, must be larger than *left*.

**size**

[int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `left`, `mode`, and `right` are all scalars. Otherwise, `np.broadcast(left, mode, right).size` samples are drawn.

**Returns**

**out**

[ndarray or scalar] Drawn samples from the parameterized triangular distribution.

**See also:**

*`Generator.triangular`*

which should be used for new code.

## Notes

The probability density function for the triangular distribution is

$$P(x;l,m,r) = \begin{cases} \frac{2(x-l)}{(r-l)(m-l)} & \text{for } l \leq x \leq m, \\ \frac{2(r-x)}{(r-l)(r-m)} & \text{for } m \leq x \leq r, \\ 0 & \text{otherwise}. \end{cases}$$

The triangular distribution is often used in ill-defined problems where the underlying distribution is not known, but some knowledge of the limits and mode exists. Often it is used in simulations.

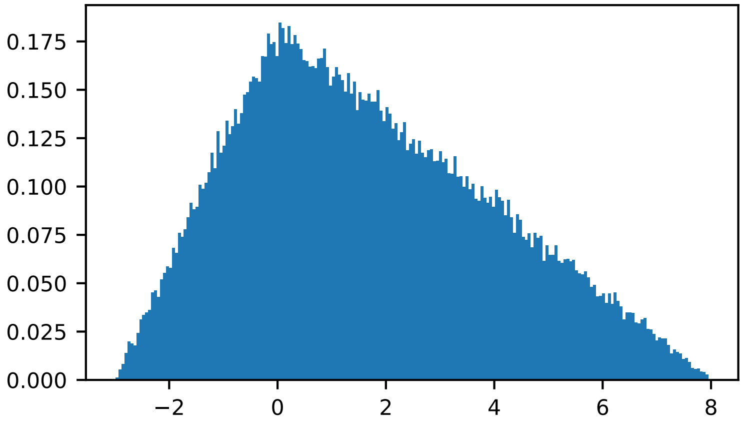
## References

[1]

## Examples

Draw values from the distribution and plot the histogram:

```
>>> import matplotlib.pyplot as plt
>>> h = plt.hist(np.random.triangular(-3, 0, 8, 100000), bins=200,
...              density=True)
>>> plt.show()
```



method

`RandomState.`**`uniform`**(*low=0.0*, *high=1.0*, *size=None*)

Draw samples from a uniform distribution.

Samples are uniformly distributed over the half-open interval `[low, high)` (includes low, but excludes high). In other words, any value within the given interval is equally likely to be drawn by *`uniform`*.

---

**Note:** New code should use the `uniform` method of a `default_rng()` instance instead; see *random-quick-start*.

---

**Parameters**

**low**

[float or array_like of floats, optional] Lower boundary of the output interval. All values generated will be greater than or equal to low. The default value is 0.

**high**

[float or array_like of floats] Upper boundary of the output interval. All values generated will be less than or equal to high. The default value is 1.0.

**size**

[int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `low` and `high` are both scalars. Otherwise, `np.broadcast(low, high).size` samples are drawn.

**Returns**

**out**

[ndarray or scalar] Drawn samples from the parameterized uniform distribution.

**See also:**

***`randint`***

Discrete uniform distribution, yielding integers.

***`random_integers`***

Discrete uniform distribution over the closed interval `[low, high]`.

***`random_sample`***

Floats uniformly distributed over `[0, 1)`.

***`random`***

Alias for *`random_sample`*.

***`rand`***

Convenience function that accepts dimensions as input, e.g., `rand(2,2)` would generate a 2-by-2 array of floats, uniformly distributed over `[0, 1)`.

***`Generator.uniform`***

which should be used for new code.

## Notes

The probability density function of the uniform distribution is

$$p(x) = \frac{1}{b - a}$$

anywhere within the interval `[a, b)`, and zero elsewhere.

When `high` == `low`, values of `low` will be returned. If `high` < `low`, the results are officially undefined and may eventually raise an error, i.e. do not rely on this function to behave when passed arguments satisfying that inequality condition. The `high` limit may be included in the returned array of floats due to floating-point rounding in the equation `low + (high-low) * random_sample()`. For example:

```
>>> x = np.float32(5*0.99999999)
>>> x
5.0
```

## Examples

Draw samples from the distribution:

```
>>> s = np.random.uniform(-1,0,1000)
```

All values are within the given interval:

```
>>> np.all(s >= -1)
True
>>> np.all(s < 0)
True
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 15, density=True)
>>> plt.plot(bins, np.ones_like(bins), linewidth=2, color='r')
>>> plt.show()
```

method

`RandomState`**`.vonmises`**(*mu*, *kappa*, *size=None*)

Draw samples from a von Mises distribution.

Samples are drawn from a von Mises distribution with specified mode (mu) and dispersion (kappa), on the interval [-pi, pi].

The von Mises distribution (also known as the circular normal distribution) is a continuous probability distribution on the unit circle. It may be thought of as the circular analogue of the normal distribution.

---

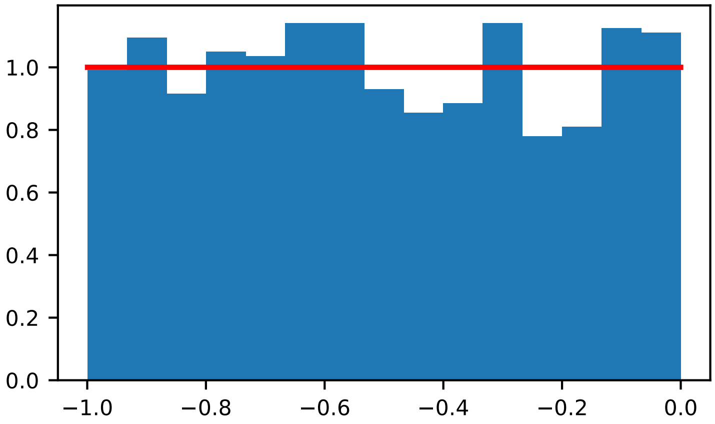
**Note:** New code should use the `vonmises` method of a `default_rng()` instance instead; see *random-quick-start*.

---

**Parameters**

**mu**

[float or array_like of floats] Mode (“center”) of the distribution.



**kappa**

[float or array_like of floats] Dispersion of the distribution, has to be >=0.

**size**

[int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `mu` and `kappa` are both scalars. Otherwise, `np.broadcast(mu, kappa).size` samples are drawn.

**Returns**

**out**

[ndarray or scalar] Drawn samples from the parameterized von Mises distribution.

**See also:**

**`scipy.stats.vonmises`**

probability density function, distribution, or cumulative density function, etc.

***`Generator.vonmises`***

which should be used for new code.

## Notes

The probability density for the von Mises distribution is

$$p(x) = \frac{e^{\kappa cos(x-\mu)}}{2\pi I_0(\kappa)},$$

where $\mu$ is the mode and $\kappa$ the dispersion, and $I_0(\kappa)$ is the modified Bessel function of order 0.

The von Mises is named for Richard Edler von Mises, who was born in Austria-Hungary, in what is now the Ukraine. He fled to the United States in 1939 and became a professor at Harvard. He worked in probability theory, aerodynamics, fluid mechanics, and philosophy of science.

### References

[1], [2]

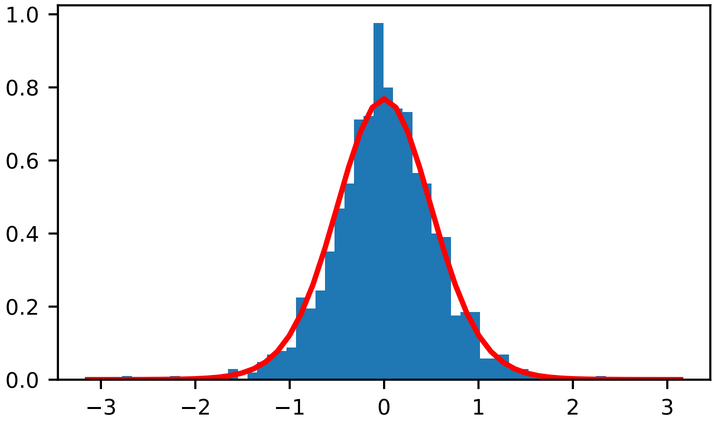
### Examples

Draw samples from the distribution:

```
>>> mu, kappa = 0.0, 4.0 # mean and dispersion
>>> s = np.random.vonmises(mu, kappa, 1000)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> from scipy.special import i0
>>> plt.hist(s, 50, density=True)
>>> x = np.linspace(-np.pi, np.pi, num=51)
>>> y = np.exp(kappa*np.cos(x-mu))/(2*np.pi*i0(kappa))
>>> plt.plot(x, y, linewidth=2, color='r')
>>> plt.show()
```



method

`RandomState.`**`wald`**(*mean*, *scale*, *size=None*)

Draw samples from a Wald, or inverse Gaussian, distribution.

As the scale approaches infinity, the distribution becomes more like a Gaussian. Some references claim that the Wald is an inverse Gaussian with mean equal to 1, but this is by no means universal.

The inverse Gaussian distribution was first studied in relationship to Brownian motion. In 1956 M.C.K. Tweedie used the name inverse Gaussian because there is an inverse relationship between the time to cover a unit distance and distance covered in unit time.

**Note:** New code should use the `wald` method of a `default_rng()` instance instead; see *random-quick-start*.

**Parameters**

**mean**

[float or array_like of floats] Distribution mean, must be > 0.

**scale**

[float or array_like of floats] Scale parameter, must be > 0.

**size**

[int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `mean` and `scale` are both scalars. Otherwise, `np.broadcast(mean, scale).size` samples are drawn.

**Returns**

**out**

[ndarray or scalar] Drawn samples from the parameterized Wald distribution.

**See also:**

***`Generator.wald`***

which should be used for new code.

## Notes

The probability density function for the Wald distribution is

$$P(x;mean,scale) = \sqrt{\frac{scale}{2\pi x^3}}e^{\frac{-scale(x-mean)^2}{2\cdot mean^2 x}}$$

As noted above the inverse Gaussian distribution first arise from attempts to model Brownian motion. It is also a competitor to the Weibull for use in reliability modeling and modeling stock returns and interest rate processes.

## References

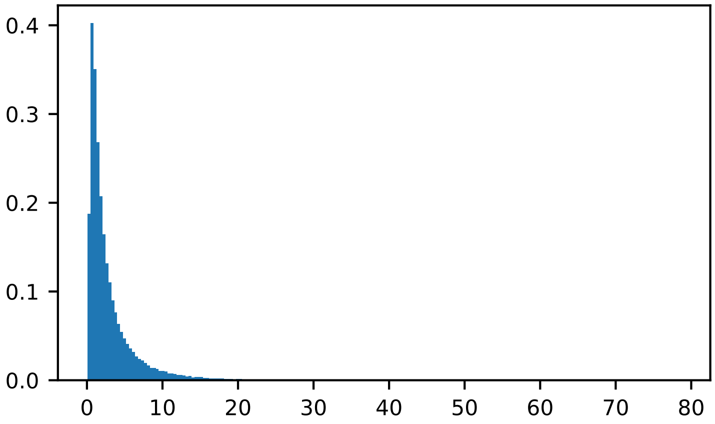
[1], [2], [3]

## Examples

Draw values from the distribution and plot the histogram:

```
>>> import matplotlib.pyplot as plt
>>> h = plt.hist(np.random.wald(3, 2, 100000), bins=200, density=True)
>>> plt.show()
```

method



`RandomState.`**`weibull`**`(`*a*, *size=None*`)`

Draw samples from a Weibull distribution.

Draw samples from a 1-parameter Weibull distribution with the given shape parameter *a*.

$$X = (-ln(U))^{1/a}$$

Here, U is drawn from the uniform distribution over (0,1].

The more common 2-parameter Weibull, including a scale parameter $\lambda$ is just $X = \lambda(-ln(U))^{1/a}$.

**Note:** New code should use the `weibull` method of a `default_rng()` instance instead; see *random-quick-start*.

**Parameters**

**a**

[float or array_like of floats] Shape parameter of the distribution. Must be nonnegative.

**size**

[int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `a` is a scalar. Otherwise, `np.array(a).size` samples are drawn.

**Returns**

**out**

[ndarray or scalar] Drawn samples from the parameterized Weibull distribution.

**See also:**

`scipy.stats.weibull_max`, `scipy.stats.weibull_min`, `scipy.stats.genextreme`, *`gumbel`*

*Generator.weibull* which should be used for new code.

### Notes

The Weibull (or Type III asymptotic extreme value distribution for smallest values, SEV Type III, or Rosin-Rammler distribution) is one of a class of Generalized Extreme Value (GEV) distributions used in modeling extreme value problems. This class includes the Gumbel and Frechet distributions.

The probability density for the Weibull distribution is

$$p(x) = \frac{a}{\lambda}(\frac{x}{\lambda})^{a-1}e^{-(x/\lambda)^a},$$

where $a$ is the shape and $\lambda$ the scale.

The function has its peak (the mode) at $\lambda(\frac{a-1}{a})^{1/a}$.

When `a = 1`, the Weibull distribution reduces to the exponential distribution.

### References

[1], [2], [3]

### Examples

Draw samples from the distribution:

```
>>> a = 5. # shape
>>> s = np.random.weibull(a, 1000)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> x = np.arange(1,100.)/50.
>>> def weib(x,n,a):
...     return (a / n) * (x / n)**(a - 1) * np.exp(-(x / n)**a)
```

```
>>> count, bins, ignored = plt.hist(np.random.weibull(5.,1000))
>>> x = np.arange(1,100.)/50.
>>> scale = count.max()/weib(x, 1., 5.).max()
>>> plt.plot(x, weib(x, 1., 5.)*scale)
>>> plt.show()
```

method

`RandomState.`**`zipf`**`(`*a*, *size=None*`)`

Draw samples from a Zipf distribution.

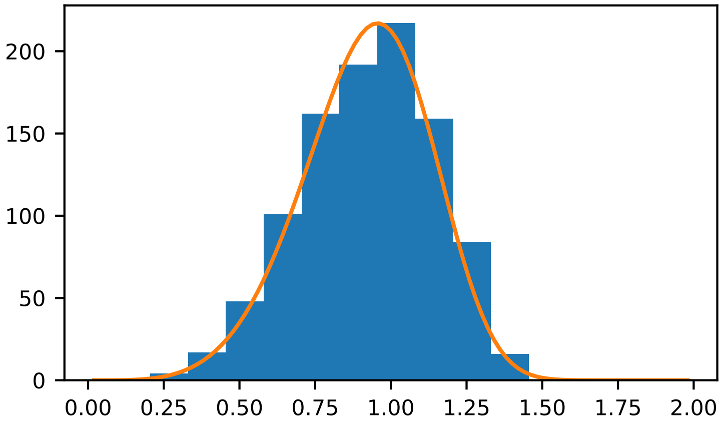
Samples are drawn from a Zipf distribution with specified parameter *a* > 1.

The Zipf distribution (also known as the zeta distribution) is a continuous probability distribution that satisfies Zipf's law: the frequency of an item is inversely proportional to its rank in a frequency table.

---

**Note:** New code should use the `zipf` method of a `default_rng()` instance instead; see *random-quick-start*.

---



**Parameters**

**a**

[float or array_like of floats] Distribution parameter. Must be greater than 1.

**size**

[int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `a` is a scalar. Otherwise, `np.array(a).size` samples are drawn.

**Returns**

**out**

[ndarray or scalar] Drawn samples from the parameterized Zipf distribution.

**See also:**

**`scipy.stats.zipf`**

probability density function, distribution, or cumulative density function, etc.

*`Generator.zipf`*

which should be used for new code.

### Notes

The probability density for the Zipf distribution is

$$p(x) = \frac{x^{-a}}{\zeta(a)},$$

where $\zeta$ is the Riemann Zeta function.

It is named for the American linguist George Kingsley Zipf, who noted that the frequency of any word in a sample of a language is inversely proportional to its rank in the frequency table.

### References

[1]

### Examples

Draw samples from the distribution:

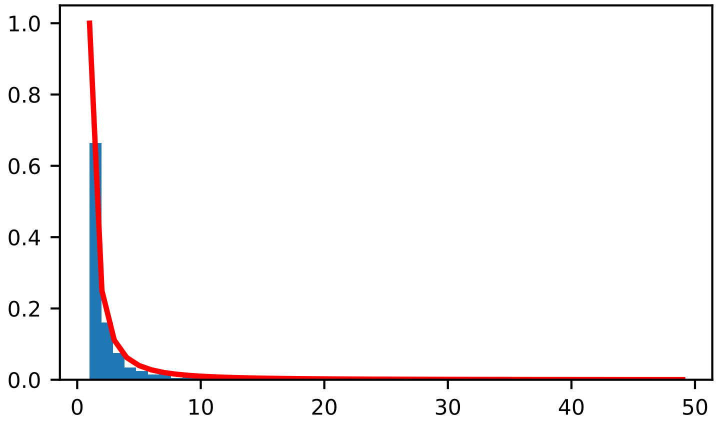
```
>>> a = 2. # parameter
>>> s = np.random.zipf(a, 1000)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> from scipy import special
```

Truncate s values at 50 so plot is interesting:

```
>>> count, bins, ignored = plt.hist(s[s<50], 50, density=True)
>>> x = np.arange(1., 50.)
>>> y = x**(-a) / special.zetac(a)
>>> plt.plot(x, y/max(y), linewidth=2, color='r')
>>> plt.show()
```



### Functions in `numpy.random`

Many of the RandomState methods above are exported as functions in `numpy.random` This usage is discouraged, as it is implemented via a global `RandomState` instance which is not advised on two counts:

- It uses global state, which means results will change as the code changes
- It uses a `RandomState` rather than the more modern `Generator`.

For backward compatible legacy reasons, we cannot change this. See *random-quick-start*.

| | |
|---|---|
| `beta`(a, b[, size]) | Draw samples from a Beta distribution. |
| `binomial`(n, p[, size]) | Draw samples from a binomial distribution. |
| `bytes`(length) | Return random bytes. |
| `chisquare`(df[, size]) | Draw samples from a chi-square distribution. |
| `choice`(a[, size, replace, p]) | Generates a random sample from a given 1-D array |
| `dirichlet`(alpha[, size]) | Draw samples from the Dirichlet distribution. |
| `exponential`([scale, size]) | Draw samples from an exponential distribution. |
| `f`(dfnum, dfden[, size]) | Draw samples from an F distribution. |
| `gamma`(shape[, scale, size]) | Draw samples from a Gamma distribution. |
| `geometric`(p[, size]) | Draw samples from the geometric distribution. |
| `get_state`() | Return a tuple representing the internal state of the generator. |
| `gumbel`([loc, scale, size]) | Draw samples from a Gumbel distribution. |
| `hypergeometric`(ngood, nbad, nsample[, size]) | Draw samples from a Hypergeometric distribution. |
| `laplace`([loc, scale, size]) | Draw samples from the Laplace or double exponential distribution with specified location (or mean) and scale (decay). |
| `logistic`([loc, scale, size]) | Draw samples from a logistic distribution. |
| `lognormal`([mean, sigma, size]) | Draw samples from a log-normal distribution. |
| `logseries`(p[, size]) | Draw samples from a logarithmic series distribution. |
| `multinomial`(n, pvals[, size]) | Draw samples from a multinomial distribution. |
| `multivariate_normal`(mean, cov[, size, …]) | Draw random samples from a multivariate normal distribution. |
| `negative_binomial`(n, p[, size]) | Draw samples from a negative binomial distribution. |
| `noncentral_chisquare`(df, nonc[, size]) | Draw samples from a noncentral chi-square distribution. |
| `noncentral_f`(dfnum, dfden, nonc[, size]) | Draw samples from the noncentral F distribution. |
| `normal`([loc, scale, size]) | Draw random samples from a normal (Gaussian) distribution. |
| `pareto`(a[, size]) | Draw samples from a Pareto II or Lomax distribution with specified shape. |
| `permutation`(x) | Randomly permute a sequence, or return a permuted range. |
| `poisson`([lam, size]) | Draw samples from a Poisson distribution. |
| `power`(a[, size]) | Draws samples in [0, 1] from a power distribution with positive exponent a - 1. |
| `rand`(d0, d1, …, dn) | Random values in a given shape. |
| `randint`(low[, high, size, dtype]) | Return random integers from *low* (inclusive) to *high* (exclusive). |
| `randn`(d0, d1, …, dn) | Return a sample (or samples) from the "standard normal" distribution. |
| `random`([size]) | Return random floats in the half-open interval [0.0, 1.0). |
| `random_integers`(low[, high, size]) | Random integers of type *np.int_* between *low* and *high*, inclusive. |

Continued on next page

Table 151 – continued from previous page

| | |
|---|---|
| *random_sample*([size]) | Return random floats in the half-open interval [0.0, 1.0). |
| *ranf*() | This is an alias of *random_sample*. |
| *rayleigh*([scale, size]) | Draw samples from a Rayleigh distribution. |
| *sample*() | This is an alias of *random_sample*. |
| *seed*(self[, seed]) | Reseed a legacy MT19937 BitGenerator |
| *set_state*(state) | Set the internal state of the generator from a tuple. |
| *shuffle*(x) | Modify a sequence in-place by shuffling its contents. |
| *standard_cauchy*([size]) | Draw samples from a standard Cauchy distribution with mode = 0. |
| *standard_exponential*([size]) | Draw samples from the standard exponential distribution. |
| *standard_gamma*(shape[, size]) | Draw samples from a standard Gamma distribution. |
| *standard_normal*([size]) | Draw samples from a standard Normal distribution (mean=0, stdev=1). |
| *standard_t*(df[, size]) | Draw samples from a standard Student's t distribution with *df* degrees of freedom. |
| *triangular*(left, mode, right[, size]) | Draw samples from the triangular distribution over the interval `[left, right]`. |
| *uniform*([low, high, size]) | Draw samples from a uniform distribution. |
| *vonmises*(mu, kappa[, size]) | Draw samples from a von Mises distribution. |
| *wald*(mean, scale[, size]) | Draw samples from a Wald, or inverse Gaussian, distribution. |
| *weibull*(a[, size]) | Draw samples from a Weibull distribution. |
| *zipf*(a[, size]) | Draw samples from a Zipf distribution. |

`numpy.random.`**`beta`**(*a*, *b*, *size=None*)

Draw samples from a Beta distribution.

The Beta distribution is a special case of the Dirichlet distribution, and is related to the Gamma distribution. It has the probability distribution function

$$f(x; a,b) = \frac{1}{B(\alpha, \beta)} x^{\alpha - 1} (1 - x)^{\beta - 1},$$

where the normalization, B, is the beta function,

$$B(\alpha, \beta) = \int_0^1 t^{\alpha - 1} (1 - t)^{\beta - 1} dt.$$

It is often seen in Bayesian inference and order statistics.

**Note:** New code should use the `beta` method of a `default_rng()` instance instead; see *random-quick-start*.

**Parameters**

**a**

[float or array_like of floats] Alpha, positive (>0).

**b**

[float or array_like of floats] Beta, positive (>0).

**size**

[int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `a` and `b` are both scalars. Otherwise, `np.broadcast(a, b).size` samples are drawn.

**Returns**

**out**

[ndarray or scalar] Drawn samples from the parameterized beta distribution.

**See also:**

*`Generator.beta`*

which should be used for new code.

`numpy.random.`**`binomial`**`(`*n*, *p*, *size=None*`)`

Draw samples from a binomial distribution.

Samples are drawn from a binomial distribution with specified parameters, n trials and p probability of success where n an integer >= 0 and p is in the interval [0,1]. (n may be input as a float, but it is truncated to an integer in use)

**Note:** New code should use the `binomial` method of a `default_rng()` instance instead; see *random-quick-start*.

**Parameters**

**n**

[int or array_like of ints] Parameter of the distribution, >= 0. Floats are also accepted, but they will be truncated to integers.

**p**

[float or array_like of floats] Parameter of the distribution, >= 0 and <=1.

**size**

[int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `n` and `p` are both scalars. Otherwise, `np.broadcast(n, p).size` samples are drawn.

**Returns**

**out**

[ndarray or scalar] Drawn samples from the parameterized binomial distribution, where each sample is equal to the number of successes over the n trials.

**See also:**

**`scipy.stats.binom`**

probability density function, distribution or cumulative density function, etc.

`Generator.binomial`

which should be used for new code.

### Notes

The probability density for the binomial distribution is

$$P(N) = \binom{n}{N} p^N (1-p)^{n-N},$$

where $n$ is the number of trials, $p$ is the probability of success, and $N$ is the number of successes.

When estimating the standard error of a proportion in a population by using a random sample, the normal distribution works well unless the product p*n <=5, where p = population proportion estimate, and n = number of samples, in which case the binomial distribution is used instead. For example, a sample of 15 people shows 4 who are left handed, and 11 who are right handed. Then p = 4/15 = 27%. 0.27*15 = 4, so the binomial distribution should be used in this case.

### References

[1], [2], [3], [4], [5]

### Examples

Draw samples from the distribution:

```
>>> n, p = 10, .5  # number of trials, probability of each trial
>>> s = np.random.binomial(n, p, 1000)
# result of flipping a coin 10 times, tested 1000 times.
```

A real world example. A company drills 9 wild-cat oil exploration wells, each with an estimated probability of success of 0.1. All nine wells fail. What is the probability of that happening?

Let's do 20,000 trials of the model, and count the number that generate zero positive results.

```
>>> sum(np.random.binomial(9, 0.1, 20000) == 0)/20000.
# answer = 0.38885, or 38%.
```

`numpy.random.`**`bytes`**(*length*)

Return random bytes.

**Note:** New code should use the `bytes` method of a `default_rng()` instance instead; see *random-quick-start*.

**Parameters**

**length**

[int] Number of random bytes.

**Returns**

**out**

[str] String of length *length*.

**See also:**

**`Generator.bytes`**

which should be used for new code.

### Examples

```
>>> np.random.bytes(10)
' eh\x85\x022SZ\xbf\xa4' #random
```

`numpy.random.`**`chisquare`**`(`*df*, *size=None*`)`

Draw samples from a chi-square distribution.

When *df* independent random variables, each with standard normal distributions (mean 0, variance 1), are squared and summed, the resulting distribution is chi-square (see Notes). This distribution is often used in hypothesis testing.

---

**Note:** New code should use the `chisquare` method of a `default_rng()` instance instead; see *random-quick-start*.

---

**Parameters**

**df**

[float or array_like of floats] Number of degrees of freedom, must be > 0.

**size**

[int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `df` is a scalar. Otherwise, `np.array(df).size` samples are drawn.

**Returns**

**out**

[ndarray or scalar] Drawn samples from the parameterized chi-square distribution.

**Raises**

**ValueError**

When *df* <= 0 or when an inappropriate *size* (e.g. `size=-1`) is given.

**See also:**

**`Generator.chisquare`**

which should be used for new code.

### Notes

The variable obtained by summing the squares of *df* independent, standard normally distributed random variables:

$$Q = \sum_{i=0}^{\mathtt{df}} X_i^2$$

is chi-square distributed, denoted

$$Q \sim \chi^2_k.$$

The probability density function of the chi-squared distribution is

$$p(x) = \frac{(1/2)^{k/2}}{\Gamma(k/2)} x^{k/2 - 1} e^{-x/2},$$

where $\Gamma$ is the gamma function,

$$\Gamma(x) = \int_0^{-\infty} t^{x-1} e^{-t} dt.$$

### References

[1]

### Examples

```
>>> np.random.chisquare(2,4)
array([ 1.89920014,  9.00867716,  3.13710533,  5.62318272]) # random
```

`numpy.random.`**`choice`**(*a*, *size=None*, *replace=True*, *p=None*)

Generates a random sample from a given 1-D array

New in version 1.7.0.

**Note:** New code should use the `choice` method of a `default_rng()` instance instead; see *random-quick-start*.

**Parameters**

**a**

[1-D array-like or int] If an ndarray, a random sample is generated from its elements. If an int, the random sample is generated as if a were np.arange(a)

**size**

[int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. Default is None, in which case a single value is returned.

**replace**

[boolean, optional] Whether the sample is with or without replacement

**p**

[1-D array-like, optional] The probabilities associated with each entry in a. If not given the sample assumes a uniform distribution over all entries in a.

**Returns**

**samples**

[single item or ndarray] The generated random samples

**Raises**

**ValueError**

If a is an int and less than zero, if a or p are not 1-dimensional, if a is an array-like of size 0, if p is not a vector of probabilities, if a and p have different lengths, or if replace=False and the sample size is greater than the population size

**See also:**

`randint`, `shuffle`, `permutation`

**`Generator.choice`**

which should be used in new code

## Notes

Sampling random rows from a 2-D array is not possible with this function, but is possible with `Generator.choice` through its `axis` keyword.

## Examples

Generate a uniform random sample from np.arange(5) of size 3:

```
>>> np.random.choice(5, 3)
array([0, 3, 4]) # random
>>> #This is equivalent to np.random.randint(0,5,3)
```

Generate a non-uniform random sample from np.arange(5) of size 3:

```
>>> np.random.choice(5, 3, p=[0.1, 0, 0.3, 0.6, 0])
array([3, 3, 0]) # random
```

Generate a uniform random sample from np.arange(5) of size 3 without replacement:

```
>>> np.random.choice(5, 3, replace=False)
array([3,1,0]) # random
>>> #This is equivalent to np.random.permutation(np.arange(5))[:3]
```

Generate a non-uniform random sample from np.arange(5) of size 3 without replacement:

```
>>> np.random.choice(5, 3, replace=False, p=[0.1, 0, 0.3, 0.6, 0])
array([2, 3, 0]) # random
```

Any of the above can be repeated with an arbitrary array-like instead of just integers. For instance:

```
>>> aa_milne_arr = ['pooh', 'rabbit', 'piglet', 'Christopher']
>>> np.random.choice(aa_milne_arr, 5, p=[0.5, 0.1, 0.1, 0.3])
array(['pooh', 'pooh', 'pooh', 'Christopher', 'piglet'], # random
      dtype='<U11')
```

`numpy.random.`**`dirichlet`**(*alpha*, *size=None*)

Draw samples from the Dirichlet distribution.

Draw *size* samples of dimension k from a Dirichlet distribution. A Dirichlet-distributed random variable can be seen as a multivariate generalization of a Beta distribution. The Dirichlet distribution is a conjugate prior of a multinomial distribution in Bayesian inference.

**Note:** New code should use the `dirichlet` method of a `default_rng()` instance instead; see *random-quick-start*.

**Parameters**

**alpha**

[sequence of floats, length k] Parameter of the distribution (length `k` for sample of length `k`).

**size**

[int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n)`, then `m * n * k` samples are drawn. Default is None, in which case a vector of length `k` is returned.

**Returns**

**samples**

[ndarray,] The drawn samples, of shape `(size, k)`.

**Raises**

**ValueError**

If any value in `alpha` is less than or equal to zero

**See also:**

**`Generator.dirichlet`**

which should be used for new code.

## Notes

The Dirichlet distribution is a distribution over vectors $x$ that fulfil the conditions $x_i > 0$ and $\sum_{i=1}^{k} x_i = 1$.

The probability density function $p$ of a Dirichlet-distributed random vector $X$ is proportional to

$$p(x) \propto \prod_{i=1}^{k} x_i^{\alpha_i - 1},$$

where $\alpha$ is a vector containing the positive concentration parameters.

The method uses the following property for computation: let $Y$ be a random vector which has components that follow a standard gamma distribution, then $X = \frac{1}{\sum_{i=1}^{k} Y_i} Y$ is Dirichlet-distributed

### References

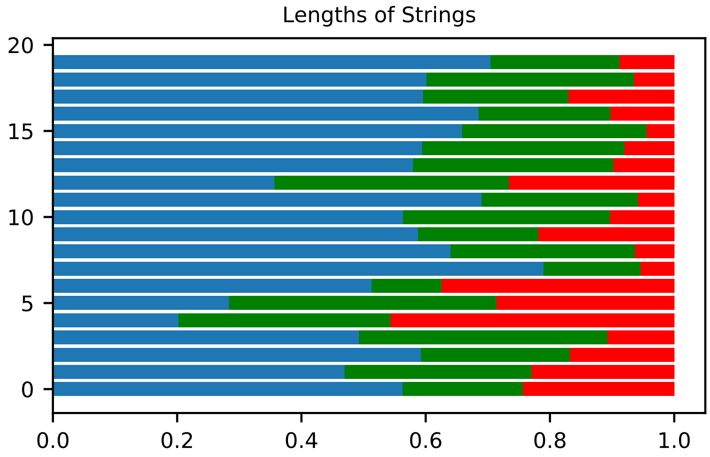
[1], [2]

### Examples

Taking an example cited in Wikipedia, this distribution can be used if one wanted to cut strings (each of initial length 1.0) into K pieces with different lengths, where each piece had, on average, a designated average length, but allowing some variation in the relative sizes of the pieces.

```
>>> s = np.random.dirichlet((10, 5, 3), 20).transpose()
```

```
>>> import matplotlib.pyplot as plt
>>> plt.barh(range(20), s[0])
>>> plt.barh(range(20), s[1], left=s[0], color='g')
>>> plt.barh(range(20), s[2], left=s[0]+s[1], color='r')
>>> plt.title("Lengths of Strings")
```



`numpy.random.`**`exponential`**(*scale=1.0*, *size=None*)

Draw samples from an exponential distribution.

Its probability density function is

$$f(x; \frac{1}{\beta}) = \frac{1}{\beta} \exp(-\frac{x}{\beta}),$$

for `x > 0` and 0 elsewhere. $\beta$ is the scale parameter, which is the inverse of the rate parameter $\lambda = 1/\beta$. The rate parameter is an alternative, widely used parameterization of the exponential distribution [3].

The exponential distribution is a continuous analogue of the geometric distribution. It describes many common situations, such as the size of raindrops measured over many rainstorms [1], or the time between page requests to Wikipedia [2].

---

**Note:** New code should use the `exponential` method of a `default_rng()` instance instead; see *random-quick-start*.

---

**Parameters**

**scale**

[float or array_like of floats] The scale parameter, $\beta = 1/\lambda$. Must be non-negative.

**size**

[int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `scale` is a scalar. Otherwise, `np.array(scale).size` samples are drawn.

**Returns**

**out**

[ndarray or scalar] Drawn samples from the parameterized exponential distribution.

**See also:**

***`Generator.exponential`***

which should be used for new code.

## References

[1], [2], [3]

`numpy.random.`**`f`**(*dfnum*, *dfden*, *size=None*)

Draw samples from an F distribution.

Samples are drawn from an F distribution with specified parameters, *dfnum* (degrees of freedom in numerator) and *dfden* (degrees of freedom in denominator), where both parameters must be greater than zero.

The random variate of the F distribution (also known as the Fisher distribution) is a continuous probability distribution that arises in ANOVA tests, and is the ratio of two chi-square variates.

---

**Note:** New code should use the `f` method of a `default_rng()` instance instead; see *random-quick-start*.

---

**Parameters**

**dfnum**

[float or array_like of floats] Degrees of freedom in numerator, must be > 0.

**dfden**

[float or array_like of float] Degrees of freedom in denominator, must be > 0.

**size**

[int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `dfnum` and `dfden` are both scalars. Otherwise, `np.broadcast(dfnum, dfden).size` samples are drawn.

**Returns**

**out**

[ndarray or scalar] Drawn samples from the parameterized Fisher distribution.

**See also:**

**`scipy.stats.f`**

probability density function, distribution or cumulative density function, etc.

***`Generator.f`***

which should be used for new code.

### Notes

The F statistic is used to compare in-group variances to between-group variances. Calculating the distribution depends on the sampling, and so it is a function of the respective degrees of freedom in the problem. The variable *dfnum* is the number of samples minus one, the between-groups degrees of freedom, while *dfden* is the within-groups degrees of freedom, the sum of the number of samples in each group minus the number of groups.

### References

[1], [2]

### Examples

An example from Glantz[1], pp 47-40:

Two groups, children of diabetics (25 people) and children from people without diabetes (25 controls). Fasting blood glucose was measured, case group had a mean value of 86.1, controls had a mean value of 82.2. Standard deviations were 2.09 and 2.49 respectively. Are these data consistent with the null hypothesis that the parents diabetic status does not affect their children's blood glucose levels? Calculating the F statistic from the data gives a value of 36.01.

Draw samples from the distribution:

```
>>> dfnum = 1. # between group degrees of freedom
>>> dfden = 48. # within groups degrees of freedom
>>> s = np.random.f(dfnum, dfden, 1000)
```

The lower bound for the top 1% of the samples is :

```
>>> np.sort(s)[-10]
7.61988120985 # random
```

So there is about a 1% chance that the F statistic will exceed 7.62, the measured value is 36, so the null hypothesis is rejected at the 1% level.

`numpy.random.`**`gamma`**`(`*shape*, *scale=1.0*, *size=None*`)`

Draw samples from a Gamma distribution.

Samples are drawn from a Gamma distribution with specified parameters, *shape* (sometimes designated “k”) and *scale* (sometimes designated “theta”), where both parameters are > 0.

---

**Note:** New code should use the `gamma` method of a `default_rng()` instance instead; see *random-quick-start*.

---

**Parameters**

**shape**

[float or array_like of floats] The shape of the gamma distribution. Must be non-negative.

**scale**

[float or array_like of floats, optional] The scale of the gamma distribution. Must be non-negative. Default is equal to 1.

**size**

[int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `shape` and `scale` are both scalars. Otherwise, `np.broadcast(shape, scale).size` samples are drawn.

**Returns**

**out**

[ndarray or scalar] Drawn samples from the parameterized gamma distribution.

**See also:**

**`scipy.stats.gamma`**

probability density function, distribution or cumulative density function, etc.

***`Generator.gamma`***

which should be used for new code.

## Notes

The probability density for the Gamma distribution is

$$p(x) = x^{k-1}\frac{e^{-x/\theta}}{\theta^k\Gamma(k)},$$

where $k$ is the shape and $\theta$ the scale, and $\Gamma$ is the Gamma function.

The Gamma distribution is often used to model the times to failure of electronic components, and arises naturally in processes for which the waiting times between Poisson distributed events are relevant.

### References

[1], [2]

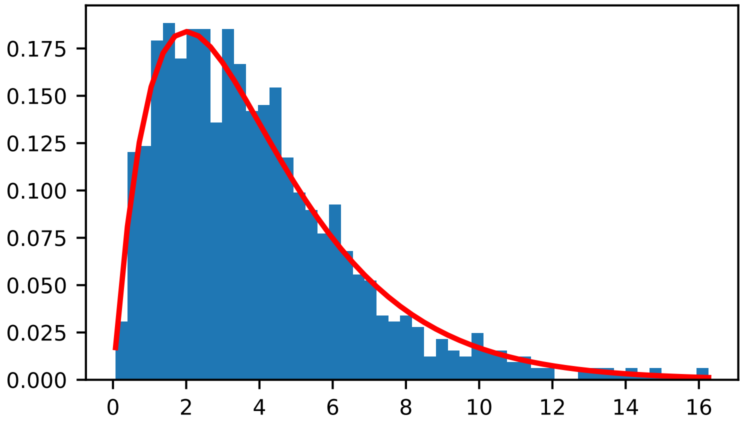
### Examples

Draw samples from the distribution:

```
>>> shape, scale = 2., 2.  # mean=4, std=2*sqrt(2)
>>> s = np.random.gamma(shape, scale, 1000)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> import scipy.special as sps
>>> count, bins, ignored = plt.hist(s, 50, density=True)
>>> y = bins**(shape-1)*(np.exp(-bins/scale) /
...                      (sps.gamma(shape)*scale**shape))
>>> plt.plot(bins, y, linewidth=2, color='r')
>>> plt.show()
```



`numpy.random.`**`geometric`**(*p*, *size=None*)

Draw samples from the geometric distribution.

Bernoulli trials are experiments with one of two outcomes: success or failure (an example of such an experiment is flipping a coin). The geometric distribution models the number of trials that must be run in order to achieve success. It is therefore supported on the positive integers, `k = 1, 2, ...`.

The probability mass function of the geometric distribution is

$$f(k) = (1 - p)^{k-1} p$$

where *p* is the probability of success of an individual trial.

**Note:** New code should use the `geometric` method of a `default_rng()` instance instead; see *random-quick-start*.

**Parameters**

**p**

[float or array_like of floats] The probability of success of an individual trial.

**size**

[int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `p` is a scalar. Otherwise, `np.array(p).size` samples are drawn.

**Returns**

**out**

[ndarray or scalar] Drawn samples from the parameterized geometric distribution.

**See also:**

***`Generator.geometric`***

which should be used for new code.

### Examples

Draw ten thousand values from the geometric distribution, with the probability of an individual success equal to 0.35:

```
>>> z = np.random.geometric(p=0.35, size=10000)
```

How many trials succeeded after a single run?

```
>>> (z == 1).sum() / 10000.
0.34889999999999999 #random
```

`numpy.random.`**`get_state`**`()`

Return a tuple representing the internal state of the generator.

For more details, see *`set_state`*.

**Parameters**

**legacy**

[bool, optional] Flag indicating to return a legacy tuple state when the BitGenerator is MT19937, instead of a dict.

**Returns**

**out**

[{tuple(str, ndarray of 624 uints, int, int, float), dict}] The returned tuple has the following items:

1. the string 'MT19937'.
2. a 1-D array of 624 unsigned integer keys.
3. an integer `pos`.

4. an integer `has_gauss`.
5. a float `cached_gaussian`.

If *legacy* is False, or the BitGenerator is not MT19937, then state is returned as a dictionary.

**See also:**

*`set_state`*

## Notes

*`set_state`* and *`get_state`* are not needed to work with any of the random distributions in NumPy. If the internal state is manually altered, the user should know exactly what he/she is doing.

`numpy.random.`**`gumbel`**`(`*loc=0.0*, *scale=1.0*, *size=None*`)`

Draw samples from a Gumbel distribution.

Draw samples from a Gumbel distribution with specified location and scale. For more information on the Gumbel distribution, see Notes and References below.

**Note:** New code should use the `gumbel` method of a `default_rng()` instance instead; see *random-quick-start*.

**Parameters**

**loc**

[float or array_like of floats, optional] The location of the mode of the distribution. Default is 0.

**scale**

[float or array_like of floats, optional] The scale parameter of the distribution. Default is 1. Must be non- negative.

**size**

[int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `loc` and `scale` are both scalars. Otherwise, `np.broadcast(loc, scale).size` samples are drawn.

**Returns**

**out**

[ndarray or scalar] Drawn samples from the parameterized Gumbel distribution.

**See also:**

`scipy.stats.gumbel_l`, `scipy.stats.gumbel_r`, `scipy.stats.genextreme`, *`weibull`*

***`Generator.gumbel`***

which should be used for new code.

### Notes

The Gumbel (or Smallest Extreme Value (SEV) or the Smallest Extreme Value Type I) distribution is one of a class of Generalized Extreme Value (GEV) distributions used in modeling extreme value problems. The Gumbel is a special case of the Extreme Value Type I distribution for maximums from distributions with "exponential-like" tails.

The probability density for the Gumbel distribution is

$$p(x) = \frac{e^{-(x - \mu)/ \beta}}{\beta} e^{ -e^{-(x - \mu)/ \beta}},$$

where $\mu$ is the mode, a location parameter, and $\beta$ is the scale parameter.

The Gumbel (named for German mathematician Emil Julius Gumbel) was used very early in the hydrology literature, for modeling the occurrence of flood events. It is also used for modeling maximum wind speed and rainfall rates. It is a "fat-tailed" distribution - the probability of an event in the tail of the distribution is larger than if one used a Gaussian, hence the surprisingly frequent occurrence of 100-year floods. Floods were initially modeled as a Gaussian process, which underestimated the frequency of extreme events.

It is one of a class of extreme value distributions, the Generalized Extreme Value (GEV) distributions, which also includes the Weibull and Frechet.

The function has a mean of $\mu + 0.57721\beta$ and a variance of $\frac{\pi^2}{6}\beta^2$.

### References

[1], [2]

### Examples

Draw samples from the distribution:

```
>>> mu, beta = 0, 0.1 # location and scale
>>> s = np.random.gumbel(mu, beta, 1000)
```

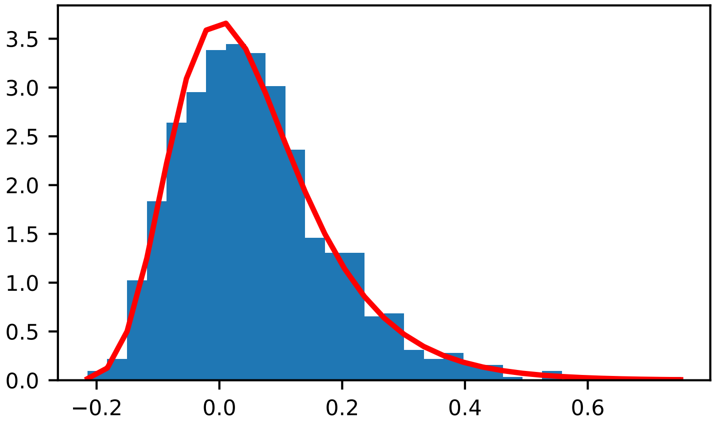
Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 30, density=True)
>>> plt.plot(bins, (1/beta)*np.exp(-(bins - mu)/beta)
...          * np.exp( -np.exp( -(bins - mu) /beta) ),
...          linewidth=2, color='r')
>>> plt.show()
```

Show how an extreme value distribution can arise from a Gaussian process and compare to a Gaussian:

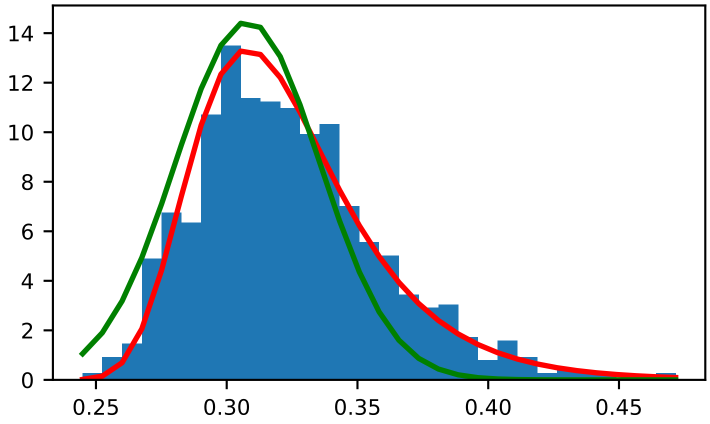
```
>>> means = []
>>> maxima = []
>>> for i in range(0,1000) :
...    a = np.random.normal(mu, beta, 1000)
...    means.append(a.mean())
...    maxima.append(a.max())
>>> count, bins, ignored = plt.hist(maxima, 30, density=True)
>>> beta = np.std(maxima) * np.sqrt(6) / np.pi
>>> mu = np.mean(maxima) - 0.57721*beta
```

(continues on next page)



(continued from previous page)

```
>>> plt.plot(bins, (1/beta)*np.exp(-(bins - mu)/beta)
...          * np.exp(-np.exp(-(bins - mu)/beta)),
...          linewidth=2, color='r')
>>> plt.plot(bins, 1/(beta * np.sqrt(2 * np.pi))
...          * np.exp(-(bins - mu)**2 / (2 * beta**2)),
...          linewidth=2, color='g')
>>> plt.show()
```



`numpy.random.`**`hypergeometric`**(*ngood*, *nbad*, *nsample*, *size=None*)

Draw samples from a Hypergeometric distribution.

Samples are drawn from a hypergeometric distribution with specified parameters, *ngood* (ways to make a good selection), *nbad* (ways to make a bad selection), and *nsample* (number of items sampled, which is less than or equal to the sum `ngood + nbad`).

---

**Note:** New code should use the `hypergeometric` method of a `default_rng()` instance instead; see *random-quick-start*.

---

**Parameters**

**ngood**

[int or array_like of ints] Number of ways to make a good selection. Must be nonnegative.

**nbad**

[int or array_like of ints] Number of ways to make a bad selection. Must be nonnegative.

**nsample**

[int or array_like of ints] Number of items sampled. Must be at least 1 and at most `ngood + nbad`.

**size**

[int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if *ngood*, *nbad*, and *nsample* are all scalars. Otherwise, `np.broadcast(ngood, nbad, nsample).size` samples are drawn.

**Returns**

**out**

[ndarray or scalar] Drawn samples from the parameterized hypergeometric distribution. Each sample is the number of good items within a randomly selected subset of size *nsample* taken from a set of *ngood* good items and *nbad* bad items.

**See also:**

**`scipy.stats.hypergeom`**

probability density function, distribution or cumulative density function, etc.

***`Generator.hypergeometric`***

which should be used for new code.

## Notes

The probability density for the Hypergeometric distribution is

$$P(x) = \frac{\binom{g}{x}\binom{b}{n-x}}{\binom{g+b}{n}},$$

where $0 \le x \le n$ and $n - b \le x \le g$

for P(x) the probability of `x` good results in the drawn sample, g = *ngood*, b = *nbad*, and n = *nsample*.

Consider an urn with black and white marbles in it, *ngood* of them are black and *nbad* are white. If you draw *nsample* balls without replacement, then the hypergeometric distribution describes the distribution of black balls in the drawn sample.

Note that this distribution is very similar to the binomial distribution, except that in this case, samples are drawn without replacement, whereas in the Binomial case samples are drawn with replacement (or the sample space is infinite). As the sample space becomes large, this distribution approaches the binomial.

### References

[1], [2], [3]

### Examples

Draw samples from the distribution:

```
>>> ngood, nbad, nsamp = 100, 2, 10
# number of good, number of bad, and number of samples
>>> s = np.random.hypergeometric(ngood, nbad, nsamp, 1000)
>>> from matplotlib.pyplot import hist
>>> hist(s)
#   note that it is very unlikely to grab both bad items
```

Suppose you have an urn with 15 white and 15 black marbles. If you pull 15 marbles at random, how likely is it that 12 or more of them are one color?

```
>>> s = np.random.hypergeometric(15, 15, 15, 100000)
>>> sum(s>=12)/100000. + sum(s<=3)/100000.
#   answer = 0.003 ... pretty unlikely!
```

`numpy.random.`**`laplace`**(*loc=0.0*, *scale=1.0*, *size=None*)

Draw samples from the Laplace or double exponential distribution with specified location (or mean) and scale (decay).

The Laplace distribution is similar to the Gaussian/normal distribution, but is sharper at the peak and has fatter tails. It represents the difference between two independent, identically distributed exponential random variables.

---

**Note:** New code should use the `laplace` method of a `default_rng()` instance instead; see *random-quick-start*.

---

**Parameters**

**loc**

[float or array_like of floats, optional] The position, $\mu$, of the distribution peak. Default is 0.

**scale**

[float or array_like of floats, optional] $\lambda$, the exponential decay. Default is 1. Must be non-negative.

**size**

[int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `loc` and `scale` are both scalars. Otherwise, `np.broadcast(loc, scale).size` samples are drawn.

**Returns**

**out**

[ndarray or scalar] Drawn samples from the parameterized Laplace distribution.

**See also:**

*`Generator.laplace`*

which should be used for new code.

### Notes

It has the probability density function

$$f(x; \mu, \lambda) = \frac{1}{2\lambda} \exp\left(-\frac{|x - \mu|}{\lambda}\right).$$

The first law of Laplace, from 1774, states that the frequency of an error can be expressed as an exponential function of the absolute magnitude of the error, which leads to the Laplace distribution. For many problems in economics and health sciences, this distribution seems to model the data better than the standard Gaussian distribution.

### References

[1], [2], [3], [4]

### Examples

Draw samples from the distribution

```
>>> loc, scale = 0., 1.
>>> s = np.random.laplace(loc, scale, 1000)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 30, density=True)
>>> x = np.arange(-8., 8., .01)
>>> pdf = np.exp(-abs(x-loc)/scale)/(2.*scale)
>>> plt.plot(x, pdf)
```

Plot Gaussian for comparison:

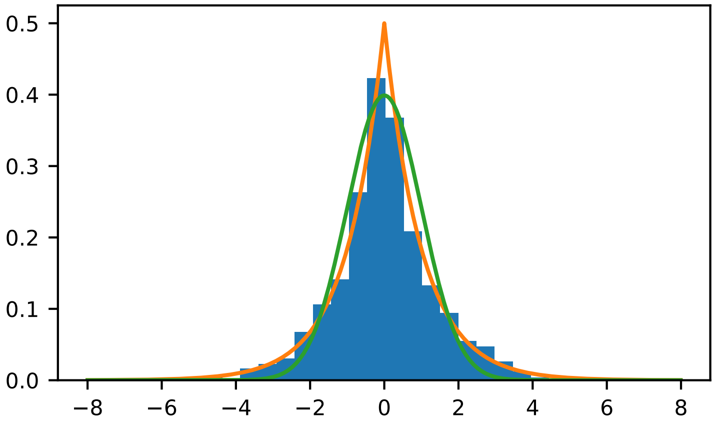
```
>>> g = (1/(scale * np.sqrt(2 * np.pi)) *
...      np.exp(-(x - loc)**2 / (2 * scale**2)))
>>> plt.plot(x,g)
```

`numpy.random.`**`logistic`**(*loc=0.0*, *scale=1.0*, *size=None*)

Draw samples from a logistic distribution.

Samples are drawn from a logistic distribution with specified parameters, loc (location or mean, also median), and scale (>0).

**Note:** New code should use the `logistic` method of a `default_rng()` instance instead; see *random-quick-start*.



**Parameters**

**loc**

[float or array_like of floats, optional] Parameter of the distribution. Default is 0.

**scale**

[float or array_like of floats, optional] Parameter of the distribution. Must be non-negative. Default is 1.

**size**

[int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `loc` and `scale` are both scalars. Otherwise, `np.broadcast(loc, scale).size` samples are drawn.

**Returns**

**out**

[ndarray or scalar] Drawn samples from the parameterized logistic distribution.

**See also:**

`scipy.stats.logistic`

probability density function, distribution or cumulative density function, etc.

*`Generator.logistic`*

which should be used for new code.

## Notes

The probability density for the Logistic distribution is

$$P(x) = P(x) = \frac{e^{-(x-\mu)/s}}{s(1+e^{-(x-\mu)/s})^2},$$

where $\mu$ = location and $s$ = scale.

The Logistic distribution is used in Extreme Value problems where it can act as a mixture of Gumbel distributions, in Epidemiology, and by the World Chess Federation (FIDE) where it is used in the Elo ranking system, assuming the performance of each player is a logistically distributed random variable.

## References

[1], [2], [3]

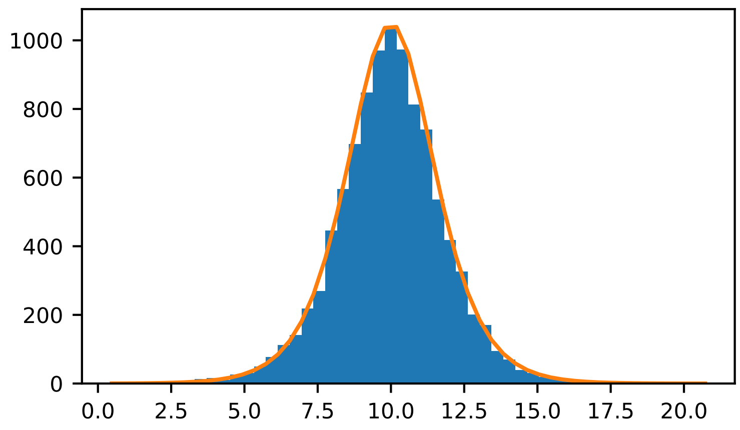
## Examples

Draw samples from the distribution:

```
>>> loc, scale = 10, 1
>>> s = np.random.logistic(loc, scale, 10000)
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, bins=50)
```

# plot against distribution

```
>>> def logist(x, loc, scale):
...     return np.exp((loc-x)/scale)/(scale*(1+np.exp((loc-x)/scale))**2)
>>> lgst_val = logist(bins, loc, scale)
>>> plt.plot(bins, lgst_val * count.max() / lgst_val.max())
>>> plt.show()
```



`numpy.random.`**`lognormal`**(*mean=0.0*, *sigma=1.0*, *size=None*)

Draw samples from a log-normal distribution.

Draw samples from a log-normal distribution with specified mean, standard deviation, and array shape. Note that the mean and standard deviation are not the values for the distribution itself, but of the underlying normal distribution it is derived from.

---

**Note:** New code should use the `lognormal` method of a `default_rng()` instance instead; see *random-quick-start*.

---

**Parameters**

**mean**

[float or array_like of floats, optional] Mean value of the underlying normal distribution. Default is 0.

**sigma**

[float or array_like of floats, optional] Standard deviation of the underlying normal distribution. Must be non-negative. Default is 1.

**size**

[int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `mean` and `sigma` are both scalars. Otherwise, `np.broadcast(mean, sigma).size` samples are drawn.

**Returns**

**out**

[ndarray or scalar] Drawn samples from the parameterized log-normal distribution.

**See also:**

**`scipy.stats.lognorm`**

probability density function, distribution, cumulative density function, etc.

***`Generator.lognormal`***

which should be used for new code.

## Notes

A variable *x* has a log-normal distribution if *log(x)* is normally distributed. The probability density function for the log-normal distribution is:

$$p(x) = \frac{1}{\sigma x \sqrt{2\pi}} e^{(-\frac{(ln(x)-\mu)^2}{2\sigma^2})}$$

where $\mu$ is the mean and $\sigma$ is the standard deviation of the normally distributed logarithm of the variable. A log-normal distribution results if a random variable is the *product* of a large number of independent, identically-distributed variables in the same way that a normal distribution results if the variable is the *sum* of a large number of independent, identically-distributed variables.

### References

[1], [2]

### Examples

Draw samples from the distribution:

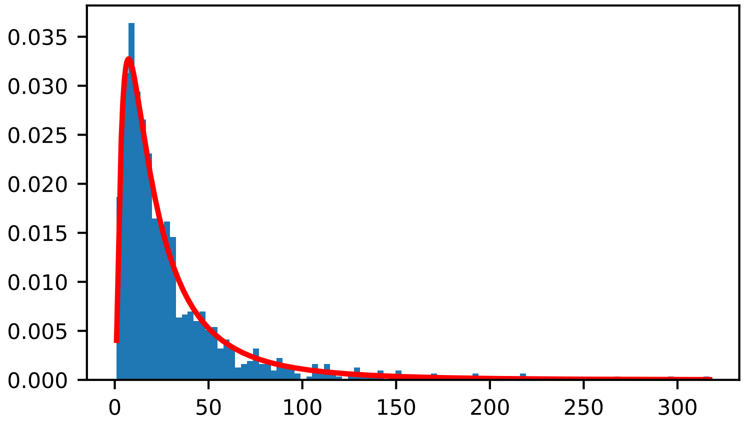
```
>>> mu, sigma = 3., 1. # mean and standard deviation
>>> s = np.random.lognormal(mu, sigma, 1000)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 100, density=True, align='mid')
```

```
>>> x = np.linspace(min(bins), max(bins), 10000)
>>> pdf = (np.exp(-(np.log(x) - mu)**2 / (2 * sigma**2))
...        / (x * sigma * np.sqrt(2 * np.pi)))
```

```
>>> plt.plot(x, pdf, linewidth=2, color='r')
>>> plt.axis('tight')
>>> plt.show()
```



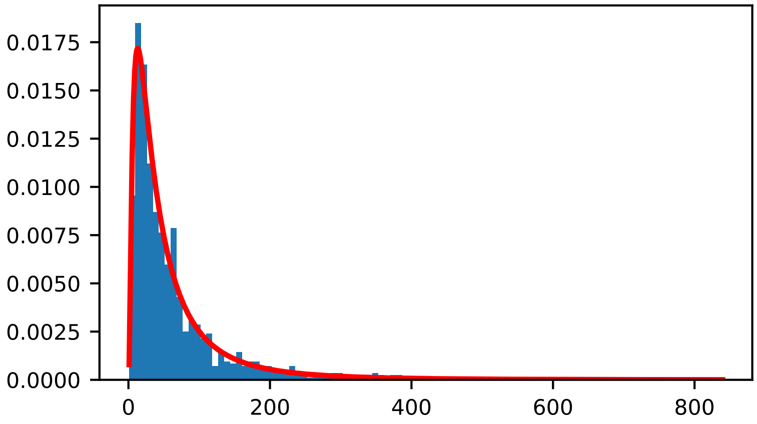
Demonstrate that taking the products of random samples from a uniform distribution can be fit well by a log-normal probability density function.

```
>>> # Generate a thousand samples: each is the product of 100 random
>>> # values, drawn from a normal distribution.
>>> b = []
>>> for i in range(1000):
...    a = 10. + np.random.standard_normal(100)
...    b.append(np.product(a))
```

```
>>> b = np.array(b) / np.min(b) # scale values to be positive
>>> count, bins, ignored = plt.hist(b, 100, density=True, align='mid')
>>> sigma = np.std(np.log(b))
>>> mu = np.mean(np.log(b))
```

```
>>> x = np.linspace(min(bins), max(bins), 10000)
>>> pdf = (np.exp(-(np.log(x) - mu)**2 / (2 * sigma**2))
...        / (x * sigma * np.sqrt(2 * np.pi)))
```

```
>>> plt.plot(x, pdf, color='r', linewidth=2)
>>> plt.show()
```



`numpy.random.`**`logseries`**(*p*, *size=None*)

Draw samples from a logarithmic series distribution.

Samples are drawn from a log series distribution with specified shape parameter, 0 < `p` < 1.

---

**Note:** New code should use the `logseries` method of a `default_rng()` instance instead; see *random-quick-start*.

---

**Parameters**

**p**

[float or array_like of floats] Shape parameter for the distribution. Must be in the range (0, 1).

**size**

[int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `p` is a scalar. Otherwise, `np.array(p).size` samples are drawn.

**Returns**

**out**

[ndarray or scalar] Drawn samples from the parameterized logarithmic series distribution.

**See also:**

`scipy.stats.logser`

probability density function, distribution or cumulative density function, etc.

*`Generator.logseries`*

which should be used for new code.

### Notes

The probability density for the Log Series distribution is

$$P(k) = \frac{-p^k}{k \ln(1-p)},$$

where p = probability.

The log series distribution is frequently used to represent species richness and occurrence, first proposed by Fisher, Corbet, and Williams in 1943 [2]. It may also be used to model the numbers of occupants seen in cars [3].

### References

[1], [2], [3], [4]

### Examples

Draw samples from the distribution:

```
>>> a = .6
>>> s = np.random.logseries(a, 10000)
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s)
```

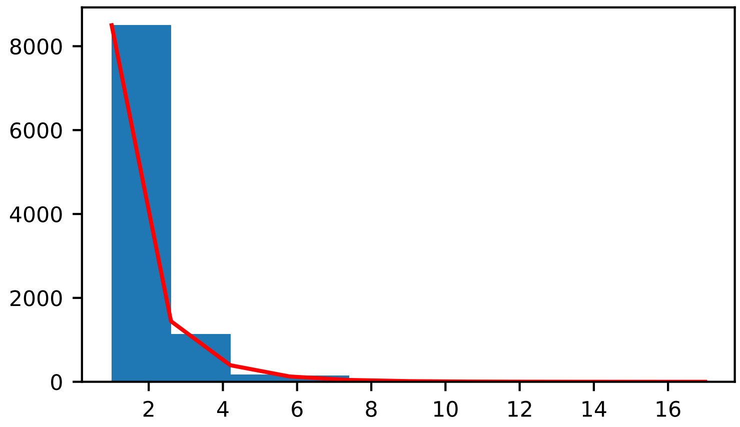
# plot against distribution

```
>>> def logseries(k, p):
...     return -p**k/(k*np.log(1-p))
>>> plt.plot(bins, logseries(bins, a)*count.max()/
...          logseries(bins, a).max(), 'r')
>>> plt.show()
```

`numpy.random.`**`multinomial`**(*n*, *pvals*, *size=None*)

Draw samples from a multinomial distribution.

The multinomial distribution is a multivariate generalization of the binomial distribution. Take an experiment with one of `p` possible outcomes. An example of such an experiment is throwing a dice, where the outcome can be 1 through 6. Each sample drawn from the distribution represents *n* such experiments. Its values, `X_i = [X_0, X_1, ..., X_p]`, represent the number of times the outcome was `i`.



**Note:** New code should use the `multinomial` method of a `default_rng()` instance instead; see *random-quick-start*.

**Parameters**

**n**

[int] Number of experiments.

**pvals**

[sequence of floats, length p] Probabilities of each of the `p` different outcomes. These must sum to 1 (however, the last element is always assumed to account for the remaining probability, as long as `sum(pvals[:-1]) <= 1)`.

**size**

[int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. Default is None, in which case a single value is returned.

**Returns**

**out**

[ndarray] The drawn samples, of shape *size*, if that was provided. If not, the shape is `(N,)`.

In other words, each entry `out[i,j,...,:]` is an N-dimensional value drawn from the distribution.

**See also:**

*`Generator.multinomial`*

which should be used for new code.

## Examples

Throw a dice 20 times:

```
>>> np.random.multinomial(20, [1/6.]*6, size=1)
array([[4, 1, 7, 5, 2, 1]]) # random
```

It landed 4 times on 1, once on 2, etc.

Now, throw the dice 20 times, and 20 times again:

```
>>> np.random.multinomial(20, [1/6.]*6, size=2)
array([[3, 4, 3, 3, 4, 3], # random
       [2, 4, 3, 4, 0, 7]])
```

For the first run, we threw 3 times 1, 4 times 2, etc. For the second, we threw 2 times 1, 4 times 2, etc.

A loaded die is more likely to land on number 6:

```
>>> np.random.multinomial(100, [1/7.]*5 + [2/7.])
array([11, 16, 14, 17, 16, 26]) # random
```

The probability inputs should be normalized. As an implementation detail, the value of the last entry is ignored and assumed to take up any leftover probability mass, but this should not be relied on. A biased coin which has twice as much weight on one side as on the other should be sampled like so:

```
>>> np.random.multinomial(100, [1.0 / 3, 2.0 / 3])  # RIGHT
array([38, 62]) # random
```

not like:

```
>>> np.random.multinomial(100, [1.0, 2.0])  # WRONG
Traceback (most recent call last):
ValueError: pvals < 0, pvals > 1 or pvals contains NaNs
```

`numpy.random.`**`multivariate_normal`**(*mean*, *cov*, *size=None*, *check_valid='warn'*, *tol=1e-8*)

Draw random samples from a multivariate normal distribution.

The multivariate normal, multinormal or Gaussian distribution is a generalization of the one-dimensional normal distribution to higher dimensions. Such a distribution is specified by its mean and covariance matrix. These parameters are analogous to the mean (average or "center") and variance (standard deviation, or "width," squared) of the one-dimensional normal distribution.

**Note:** New code should use the `multivariate_normal` method of a `default_rng()` instance instead; see *random-quick-start*.

**Parameters**

**mean**

[1-D array_like, of length N] Mean of the N-dimensional distribution.

**cov**

[2-D array_like, of shape (N, N)] Covariance matrix of the distribution. It must be symmetric and positive-semidefinite for proper sampling.

**size**

[int or tuple of ints, optional] Given a shape of, for example, `(m,n,k)`, `m*n*k` samples are generated, and packed in an *m*-by-*n*-by-*k* arrangement. Because each sample is *N*-dimensional, the output shape is `(m,n,k,N)`. If no shape is specified, a single (*N*-D) sample is returned.

**check_valid**

[{ 'warn', 'raise', 'ignore' }, optional] Behavior when the covariance matrix is not positive semidefinite.

**tol**

[float, optional] Tolerance when checking the singular values in covariance matrix. cov is cast to double before the check.

**Returns**

**out**

[ndarray] The drawn samples, of shape *size*, if that was provided. If not, the shape is `(N,)`.

In other words, each entry `out[i,j,...,:]` is an N-dimensional value drawn from the distribution.

**See also:**

***`Generator.multivariate_normal`***

which should be used for new code.

## Notes

The mean is a coordinate in N-dimensional space, which represents the location where samples are most likely to be generated. This is analogous to the peak of the bell curve for the one-dimensional or univariate normal distribution.

Covariance indicates the level to which two variables vary together. From the multivariate normal distribution, we draw N-dimensional samples, $X = [x_1, x_2, ...x_N]$. The covariance matrix element $C_{ij}$ is the covariance of $x_i$ and $x_j$. The element $C_{ii}$ is the variance of $x_i$ (i.e. its "spread").

Instead of specifying the full covariance matrix, popular approximations include:

- Spherical covariance (*cov* is a multiple of the identity matrix)
- Diagonal covariance (*cov* has non-negative elements, and only on the diagonal)

This geometrical property can be seen in two dimensions by plotting generated data-points:

```
>>> mean = [0, 0]
>>> cov = [[1, 0], [0, 100]]  # diagonal covariance
```

Diagonal covariance means that points are oriented along x or y-axis:

```
>>> import matplotlib.pyplot as plt
>>> x, y = np.random.multivariate_normal(mean, cov, 5000).T
>>> plt.plot(x, y, 'x')
>>> plt.axis('equal')
>>> plt.show()
```

Note that the covariance matrix must be positive semidefinite (a.k.a. nonnegative-definite). Otherwise, the behavior of this method is undefined and backwards compatibility is not guaranteed.

### References

[1], [2]

### Examples

```
>>> mean = (1, 2)
>>> cov = [[1, 0], [0, 1]]
>>> x = np.random.multivariate_normal(mean, cov, (3, 3))
>>> x.shape
(3, 3, 2)
```

The following is probably true, given that 0.6 is roughly twice the standard deviation:

```
>>> list((x[0,0,:] - mean) < 0.6)
[True, True] # random
```

`numpy.random.`**`negative_binomial`**`(`*n*, *p*, *size=None*`)`

Draw samples from a negative binomial distribution.

Samples are drawn from a negative binomial distribution with specified parameters, *n* successes and *p* probability of success where *n* is > 0 and *p* is in the interval [0, 1].

**Note:** New code should use the `negative_binomial` method of a `default_rng()` instance instead; see *random-quick-start*.

**Parameters**

**n**

[float or array_like of floats] Parameter of the distribution, > 0.

**p**

[float or array_like of floats] Parameter of the distribution, >= 0 and <=1.

**size**

[int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `n` and `p` are both scalars. Otherwise, `np.broadcast(n, p).size` samples are drawn.

**Returns**

**out**

[ndarray or scalar] Drawn samples from the parameterized negative binomial distribution, where each sample is equal to N, the number of failures that occurred before a total of n successes was reached.

**See also:**

***`Generator.negative_binomial`***

which should be used for new code.

### Notes

The probability mass function of the negative binomial distribution is

$$P(N; n, p) = \frac{\Gamma(N+n)}{N!\Gamma(n)} p^n (1-p)^N,$$

where $n$ is the number of successes, $p$ is the probability of success, $N + n$ is the number of trials, and $\Gamma$ is the gamma function. When $n$ is an integer, $\frac{\Gamma(N+n)}{N!\Gamma(n)} = \binom{N+n-1}{N}$, which is the more common form of this term in the the pmf. The negative binomial distribution gives the probability of N failures given n successes, with a success on the last trial.

If one throws a die repeatedly until the third time a "1" appears, then the probability distribution of the number of non-"1"s that appear before the third "1" is a negative binomial distribution.

### References

[1], [2]

### Examples

Draw samples from the distribution:

A real world example. A company drills wild-cat oil exploration wells, each with an estimated probability of success of 0.1. What is the probability of having one success for each successive well, that is what is the probability of a single success after drilling 5 wells, after 6 wells, etc.?

```
>>> s = np.random.negative_binomial(1, 0.1, 100000)
>>> for i in range(1, 11):
...     probability = sum(s<i) / 100000.
...     print(i, "wells drilled, probability of one success =", probability)
```

`numpy.random.`**`noncentral_chisquare`**(*df*, *nonc*, *size=None*)

Draw samples from a noncentral chi-square distribution.

The noncentral $\chi^2$ distribution is a generalization of the $\chi^2$ distribution.

**Note:** New code should use the `noncentral_chisquare` method of a `default_rng()` instance instead; see *random-quick-start*.

**Parameters**

**df**

[float or array_like of floats] Degrees of freedom, must be > 0.

Changed in version 1.10.0: Earlier NumPy versions required dfnum > 1.

**nonc**

[float or array_like of floats] Non-centrality, must be non-negative.

**size**

[int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `df` and `nonc` are both scalars. Otherwise, `np.broadcast(df, nonc).size` samples are drawn.

**Returns**

**out**

[ndarray or scalar] Drawn samples from the parameterized noncentral chi-square distribution.

**See also:**

***`Generator.noncentral_chisquare`***

which should be used for new code.

## Notes

The probability density function for the noncentral Chi-square distribution is

$$P(x;df,nonc) = \sum_{i=0}^{\infty} \frac{e^{-nonc/2}(nonc/2)^{i}}{i!} P_{Y_{df+2i}}(x),$$

where $Y_q$ is the Chi-square with q degrees of freedom.

## References

[1]

## Examples

Draw values from the distribution and plot the histogram

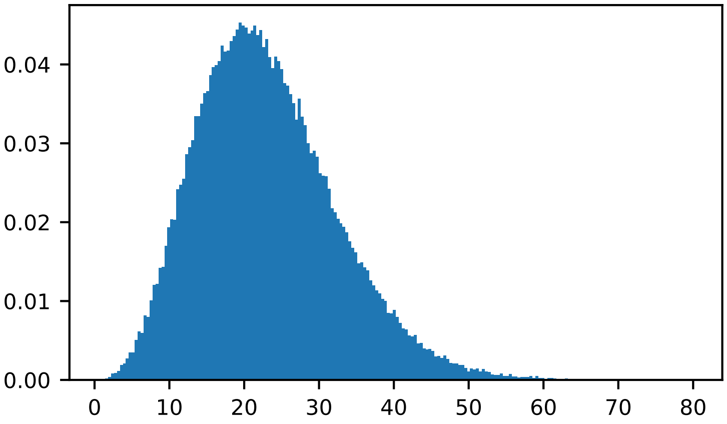
```
>>> import matplotlib.pyplot as plt
>>> values = plt.hist(np.random.noncentral_chisquare(3, 20, 100000),
...                   bins=200, density=True)
>>> plt.show()
```

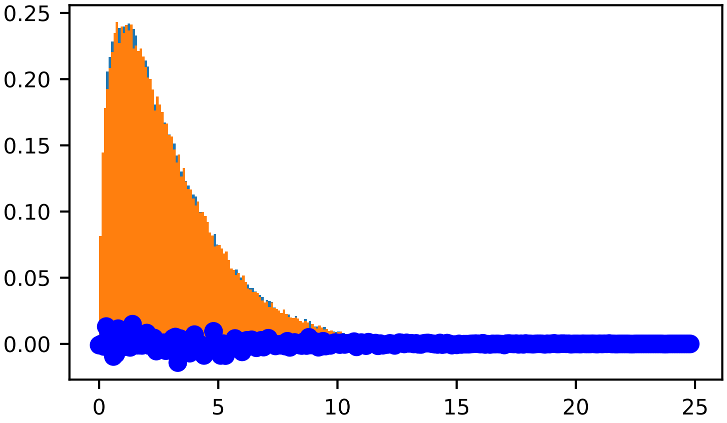
Draw values from a noncentral chisquare with very small noncentrality, and compare to a chisquare.

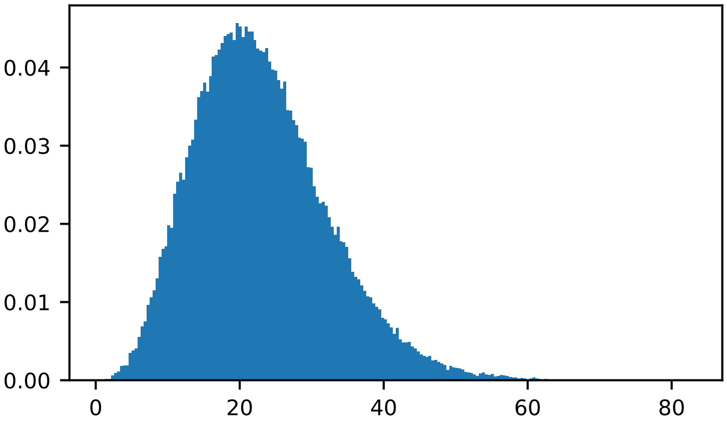
```
>>> plt.figure()
>>> values = plt.hist(np.random.noncentral_chisquare(3, .0000001, 100000),
...                   bins=np.arange(0., 25, .1), density=True)
>>> values2 = plt.hist(np.random.chisquare(3, 100000),
...                    bins=np.arange(0., 25, .1), density=True)
>>> plt.plot(values[1][0:-1], values[0]-values2[0], 'ob')
>>> plt.show()
```

Demonstrate how large values of non-centrality lead to a more symmetric distribution.

```
>>> plt.figure()
>>> values = plt.hist(np.random.noncentral_chisquare(3, 20, 100000),
...                   bins=200, density=True)
>>> plt.show()
```


0.04
0.03
0.02
0.01
0.00
0
10
20
30
40
50
60
70
80


0.25
0.20
0.15
0.10
0.05
0.00
0
5
10
15
20
25



`numpy.random.`**`noncentral_f`**`(`*dfnum*, *dfden*, *nonc*, *size=None*`)`

Draw samples from the noncentral F distribution.

Samples are drawn from an F distribution with specified parameters, *dfnum* (degrees of freedom in numerator) and *dfden* (degrees of freedom in denominator), where both parameters > 1. *nonc* is the non-centrality parameter.

**Note:** New code should use the `noncentral_f` method of a `default_rng()` instance instead; see *random-quick-start*.

**Parameters**

**dfnum**

[float or array_like of floats] Numerator degrees of freedom, must be > 0.

Changed in version 1.14.0: Earlier NumPy versions required dfnum > 1.

**dfden**

[float or array_like of floats] Denominator degrees of freedom, must be > 0.

**nonc**

[float or array_like of floats] Non-centrality parameter, the sum of the squares of the numerator means, must be >= 0.

**size**

[int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `dfnum`, `dfden`, and `nonc` are all scalars. Otherwise, `np.broadcast(dfnum, dfden, nonc).size` samples are drawn.

**Returns**

**out**

[ndarray or scalar] Drawn samples from the parameterized noncentral Fisher distribution.

**See also:**

*Generator.noncentral_f*

which should be used for new code.

### Notes

When calculating the power of an experiment (power = probability of rejecting the null hypothesis when a specific alternative is true) the non-central F statistic becomes important. When the null hypothesis is true, the F statistic follows a central F distribution. When the null hypothesis is not true, then it follows a non-central F statistic.

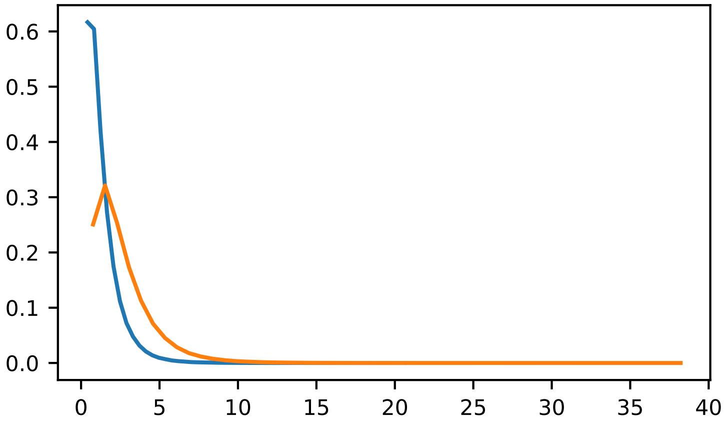
### References

[1], [2]

### Examples

In a study, testing for a specific alternative to the null hypothesis requires use of the Noncentral F distribution. We need to calculate the area in the tail of the distribution that exceeds the value of the F distribution for the null hypothesis. We'll plot the two probability distributions for comparison.

```
>>> dfnum = 3 # between group deg of freedom
>>> dfden = 20 # within groups degrees of freedom
>>> nonc = 3.0
>>> nc_vals = np.random.noncentral_f(dfnum, dfden, nonc, 1000000)
>>> NF = np.histogram(nc_vals, bins=50, density=True)
>>> c_vals = np.random.f(dfnum, dfden, 1000000)
>>> F = np.histogram(c_vals, bins=50, density=True)
>>> import matplotlib.pyplot as plt
>>> plt.plot(F[1][1:], F[0])
>>> plt.plot(NF[1][1:], NF[0])
>>> plt.show()
```



`numpy.random.normal`(*loc=0.0*, *scale=1.0*, *size=None*)

Draw random samples from a normal (Gaussian) distribution.

The probability density function of the normal distribution, first derived by De Moivre and 200 years later by both Gauss and Laplace independently [2], is often called the bell curve because of its characteristic shape (see the example below).

The normal distributions occurs often in nature. For example, it describes the commonly occurring distribution of samples influenced by a large number of tiny, random disturbances, each with its own unique distribution [2].

---

**Note:** New code should use the `normal` method of a `default_rng()` instance instead; see *random-quick-start*.

---

**Parameters**

**loc**

[float or array_like of floats] Mean ("centre") of the distribution.

**scale**

[float or array_like of floats] Standard deviation (spread or "width") of the distribution. Must be non-negative.

**size**

[int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `loc` and `scale` are both scalars. Otherwise, `np.broadcast(loc, scale).size` samples are drawn.

**Returns**

**out**

[ndarray or scalar] Drawn samples from the parameterized normal distribution.

**See also:**

**`scipy.stats.norm`**

probability density function, distribution or cumulative density function, etc.

***`Generator.normal`***

which should be used for new code.

## Notes

The probability density for the Gaussian distribution is

$$p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}},$$

where $\mu$ is the mean and $\sigma$ the standard deviation. The square of the standard deviation, $\sigma^2$, is called the variance.

The function has its peak at the mean, and its "spread" increases with the standard deviation (the function reaches 0.607 times its maximum at $x + \sigma$ and $x - \sigma$ [2]). This implies that normal is more likely to return samples lying close to the mean, rather than those far away.

### References

[1], [2]

### Examples

Draw samples from the distribution:

```
>>> mu, sigma = 0, 0.1 # mean and standard deviation
>>> s = np.random.normal(mu, sigma, 1000)
```

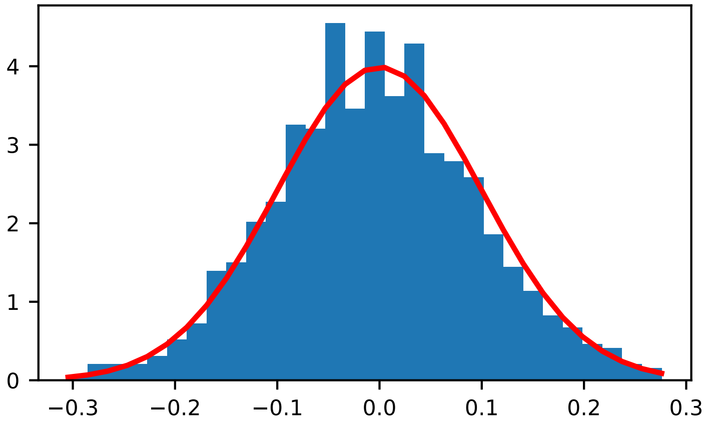
Verify the mean and the variance:

```
>>> abs(mu - np.mean(s))
0.0  # may vary
```

```
>>> abs(sigma - np.std(s, ddof=1))
0.1  # may vary
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 30, density=True)
>>> plt.plot(bins, 1/(sigma * np.sqrt(2 * np.pi)) *
...                np.exp( - (bins - mu)**2 / (2 * sigma**2) ),
...          linewidth=2, color='r')
>>> plt.show()
```



Two-by-four array of samples from N(3, 6.25):

```
>>> np.random.normal(3, 2.5, size=(2, 4))
array([[-4.49401501,  4.00950034, -1.81814867,  7.29718677],   # random
       [ 0.39924804,  4.68456316,  4.99394529,  4.84057254]])  # random
```

`numpy.random.`**`pareto`**(*a*, *size=None*)
Draw samples from a Pareto II or Lomax distribution with specified shape.

The Lomax or Pareto II distribution is a shifted Pareto distribution. The classical Pareto distribution can be obtained from the Lomax distribution by adding 1 and multiplying by the scale parameter `m` (see Notes). The smallest value of the Lomax distribution is zero while for the classical Pareto distribution it is `mu`, where the standard Pareto distribution has location `mu = 1`. Lomax can also be considered as a simplified version of the Generalized Pareto distribution (available in SciPy), with the scale set to one and the location set to zero.

The Pareto distribution must be greater than zero, and is unbounded above. It is also known as the "80-20 rule". In this distribution, 80 percent of the weights are in the lowest 20 percent of the range, while the other 20 percent fill the remaining 80 percent of the range.

**Note:** New code should use the `pareto` method of a `default_rng()` instance instead; see *random-quick-start*.

**Parameters**

**a**

[float or array_like of floats] Shape of the distribution. Must be positive.

**size**

[int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `a` is a scalar. Otherwise, `np.array(a).size` samples are drawn.

**Returns**

**out**

[ndarray or scalar] Drawn samples from the parameterized Pareto distribution.

**See also:**

**`scipy.stats.lomax`**

probability density function, distribution or cumulative density function, etc.

**`scipy.stats.genpareto`**

probability density function, distribution or cumulative density function, etc.

***`Generator.pareto`***

which should be used for new code.

## Notes

The probability density for the Pareto distribution is

$$p(x) = \frac{am^a}{x^{a+1}}$$

where $a$ is the shape and $m$ the scale.

The Pareto distribution, named after the Italian economist Vilfredo Pareto, is a power law probability distribution useful in many real world problems. Outside the field of economics it is generally referred to as the Bradford distribution. Pareto developed the distribution to describe the distribution of wealth in an economy. It has also found use in insurance, web page access statistics, oil field sizes, and many other problems, including the download frequency for projects in Sourceforge [1]. It is one of the so-called "fat-tailed" distributions.

### References

[1], [2], [3], [4]

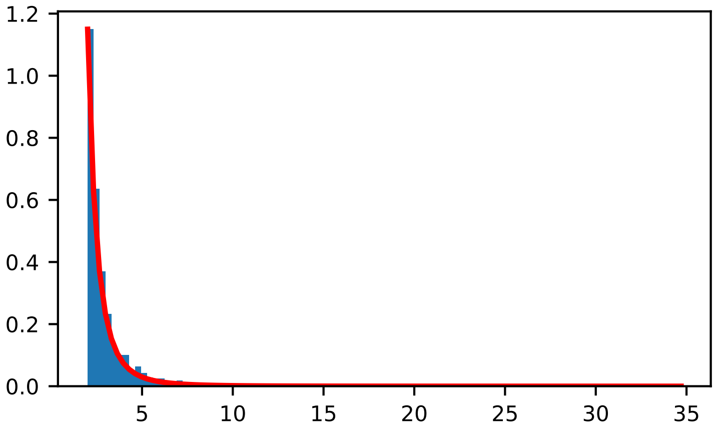
### Examples

Draw samples from the distribution:

```
>>> a, m = 3., 2.  # shape and mode
>>> s = (np.random.pareto(a, 1000) + 1) * m
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, _ = plt.hist(s, 100, density=True)
>>> fit = a*m**a / bins**(a+1)
>>> plt.plot(bins, max(count)*fit/max(fit), linewidth=2, color='r')
>>> plt.show()
```



`numpy.random.`**`permutation`**`(`*x*`)`

Randomly permute a sequence, or return a permuted range.

If *x* is a multi-dimensional array, it is only shuffled along its first index.

---

**Note:** New code should use the `permutation` method of a `default_rng()` instance instead; see *random-quick-start*.

---

**Parameters**

**x**

[int or array_like] If *x* is an integer, randomly permute `np.arange(x)`. If *x* is an array, make a copy and shuffle the elements randomly.

**Returns**

**out**

[ndarray] Permuted sequence or array range.

**See also:**

*`Generator.permutation`*

which should be used for new code.

## Examples

```
>>> np.random.permutation(10)
array([1, 7, 4, 3, 0, 9, 2, 5, 8, 6]) # random
```

```
>>> np.random.permutation([1, 4, 9, 12, 15])
array([15,  1,  9,  4, 12]) # random
```

```
>>> arr = np.arange(9).reshape((3, 3))
>>> np.random.permutation(arr)
array([[6, 7, 8], # random
       [0, 1, 2],
       [3, 4, 5]])
```

`numpy.random.`**`poisson`**(*lam=1.0*, *size=None*)

Draw samples from a Poisson distribution.

The Poisson distribution is the limit of the binomial distribution for large N.

**Note:** New code should use the `poisson` method of a `default_rng()` instance instead; see *random-quick-start*.

**Parameters**

**lam**

[float or array_like of floats] Expectation of interval, must be >= 0. A sequence of expectation intervals must be broadcastable over the requested size.

**size**

[int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `lam` is a scalar. Otherwise, `np.array(lam).size` samples are drawn.

**Returns**

**out**

[ndarray or scalar] Drawn samples from the parameterized Poisson distribution.

**See also:**

*Generator.poisson*

which should be used for new code.

### Notes

The Poisson distribution

$$f(k; \lambda)=\frac{\lambda^k e^{-\lambda}}{k!}$$

For events with an expected separation $\lambda$ the Poisson distribution $f(k; \lambda)$ describes the probability of $k$ events occurring within the observed interval $\lambda$.

Because the output is limited to the range of the C int64 type, a ValueError is raised when *lam* is within 10 sigma of the maximum representable value.

### References

[1], [2]

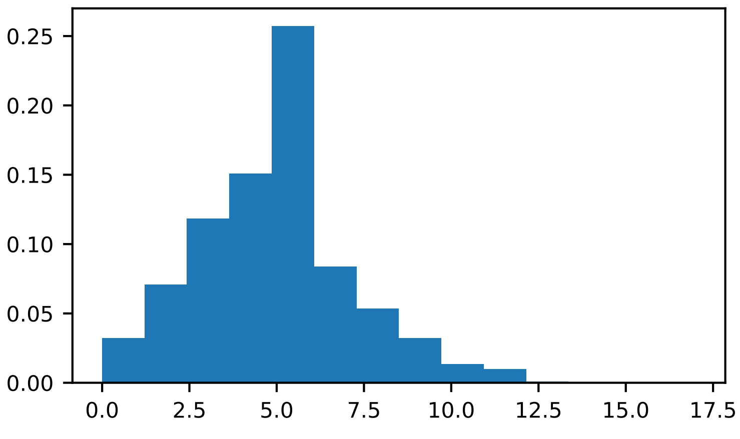
### Examples

Draw samples from the distribution:

```
>>> import numpy as np
>>> s = np.random.poisson(5, 10000)
```

Display histogram of the sample:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 14, density=True)
>>> plt.show()
```



Draw each 100 values for lambda 100 and 500:

```
>>> s = np.random.poisson(lam=(100., 500.), size=(100, 2))
```

`numpy.random.`**`power`**`(`*a*, *size=None*`)`

Draws samples in [0, 1] from a power distribution with positive exponent a - 1.

Also known as the power function distribution.

---

**Note:** New code should use the `power` method of a `default_rng()` instance instead; see *random-quick-start*.

---

**Parameters**

**a**

[float or array_like of floats] Parameter of the distribution. Must be non-negative.

**size**

[int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `a` is a scalar. Otherwise, `np.array(a).size` samples are drawn.

**Returns**

**out**

[ndarray or scalar] Drawn samples from the parameterized power distribution.

**Raises**

**ValueError**

If a < 1.

**See also:**

***`Generator.power`***

which should be used for new code.

## Notes

The probability density function is

$$P(x; a) = ax^{a-1}, 0 \le x \le 1, a > 0.$$

The power function distribution is just the inverse of the Pareto distribution. It may also be seen as a special case of the Beta distribution.

It is used, for example, in modeling the over-reporting of insurance claims.

### References

[1], [2]

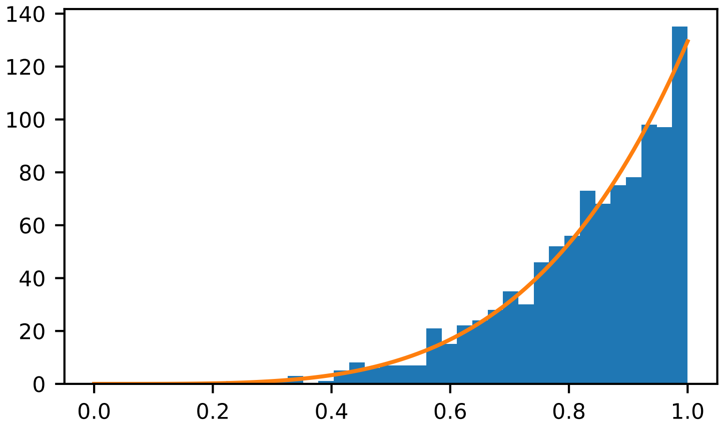
### Examples

Draw samples from the distribution:

```
>>> a = 5. # shape
>>> samples = 1000
>>> s = np.random.power(a, samples)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, bins=30)
>>> x = np.linspace(0, 1, 100)
>>> y = a*x**(a-1.)
>>> normed_y = samples*np.diff(bins)[0]*y
>>> plt.plot(x, normed_y)
>>> plt.show()
```

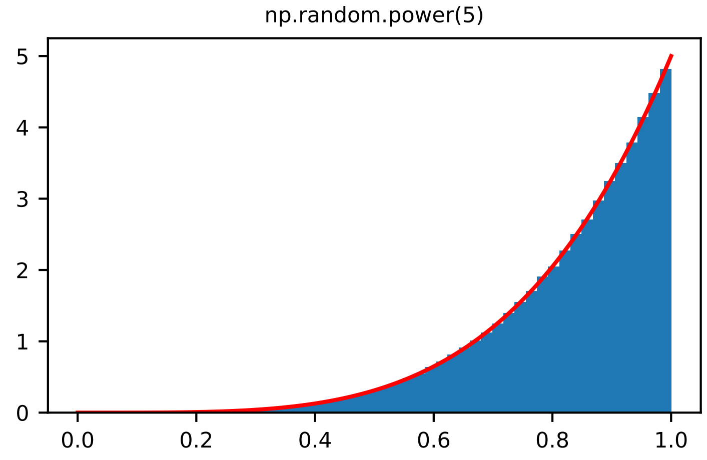


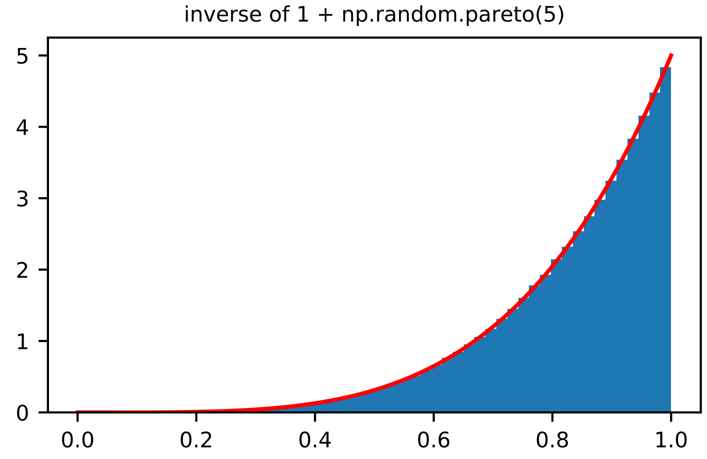
Compare the power function distribution to the inverse of the Pareto.

```
>>> from scipy import stats
>>> rvs = np.random.power(5, 1000000)
>>> rvsp = np.random.pareto(5, 1000000)
>>> xx = np.linspace(0,1,100)
>>> powpdf = stats.powerlaw.pdf(xx,5)
```

```
>>> plt.figure()
>>> plt.hist(rvs, bins=50, density=True)
>>> plt.plot(xx,powpdf,'r-')
>>> plt.title('np.random.power(5)')
>>> plt.figure()
>>> plt.hist(1./(1.+rvsp), bins=50, density=True)
>>> plt.plot(xx,powpdf,'r-')
>>> plt.title('inverse of 1 + np.random.pareto(5)')
```

```
>>> plt.figure()
>>> plt.hist(1./(1.+rvsp), bins=50, density=True)
>>> plt.plot(xx,powpdf,'r-')
>>> plt.title('inverse of stats.pareto(5)')
```



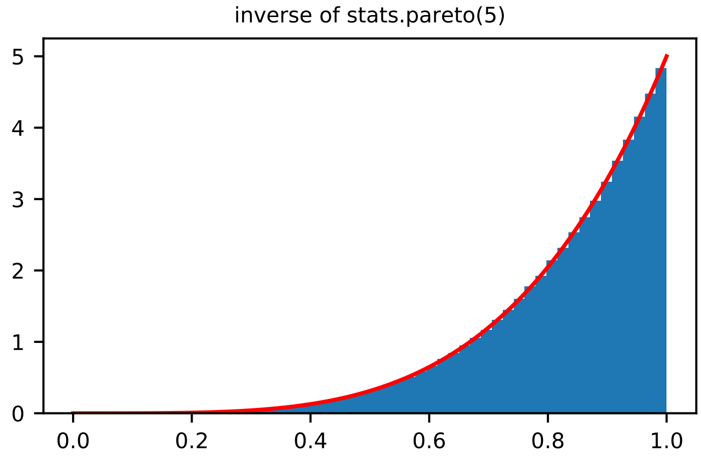


`numpy.random.`**`rand`**(*d0*, *d1*, ..., *dn*)

Random values in a given shape.

---

**Note:** This is a convenience function for users porting code from Matlab, and wraps *`random_sample`*. That



function takes a tuple to specify the size of the output, which is consistent with other NumPy functions like *`numpy.zeros`* and *`numpy.ones`*.

---

Create an array of the given shape and populate it with random samples from a uniform distribution over `[0, 1)`.

**Parameters**

**d0, d1, …, dn**

[int, optional] The dimensions of the returned array, must be non-negative. If no argument is given a single Python float is returned.

**Returns**

**out**

[ndarray, shape `(d0, d1, ..., dn)`] Random values.

**See also:**

*`random`*

## Examples

```
>>> np.random.rand(3,2)
array([[ 0.14022471,  0.96360618],  #random
       [ 0.37601032,  0.25528411],  #random
       [ 0.49313049,  0.94909878]]) #random
```

`numpy.random.`**`randint`**`(`*low*, *high=None*, *size=None*, *dtype=int*`)`

Return random integers from *low* (inclusive) to *high* (exclusive).

Return random integers from the “discrete uniform” distribution of the specified dtype in the “half-open” interval [*low*, *high*). If *high* is None (the default), then results are from [0, *low*).

**Note:** New code should use the `integers` method of a `default_rng()` instance instead; see *random-quick-start*.

**Parameters**

**low**

[int or array-like of ints] Lowest (signed) integers to be drawn from the distribution (unless `high=None`, in which case this parameter is one above the *highest* such integer).

**high**

[int or array-like of ints, optional] If provided, one above the largest (signed) integer to be drawn from the distribution (see above for behavior if `high=None`). If array-like, must contain integer values

**size**

[int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. Default is None, in which case a single value is returned.

**dtype**

[dtype, optional] Desired dtype of the result. Byteorder must be native. The default value is int.

New in version 1.11.0.

**Returns**

**out**

[int or ndarray of ints] *size*-shaped array of random integers from the appropriate distribution, or a single such random int if *size* not provided.

**See also:**

**`random_integers`**

similar to `randint`, only for the closed interval [*low*, *high*], and 1 is the lowest value if *high* is omitted.

**`Generator.integers`**

which should be used for new code.

## Examples

```
>>> np.random.randint(2, size=10)
array([1, 0, 0, 0, 1, 1, 0, 0, 1, 0]) # random
>>> np.random.randint(1, size=10)
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
```

Generate a 2 x 4 array of ints between 0 and 4, inclusive:

```
>>> np.random.randint(5, size=(2, 4))
array([[4, 0, 2, 1], # random
       [3, 2, 2, 0]])
```

Generate a 1 x 3 array with 3 different upper bounds

```
>>> np.random.randint(1, [3, 5, 10])
array([2, 2, 9]) # random
```

Generate a 1 by 3 array with 3 different lower bounds

```
>>> np.random.randint([1, 5, 7], 10)
array([9, 8, 7]) # random
```

Generate a 2 by 4 array using broadcasting with dtype of uint8

```
>>> np.random.randint([1, 3, 5, 7], [[10], [20]], dtype=np.uint8)
array([[ 8,  6,  9,  7], # random
       [ 1, 16,  9, 12]], dtype=uint8)
```

`numpy.random.`**`randn`**(*d0*, *d1*, *...*, *dn*)

Return a sample (or samples) from the "standard normal" distribution.

---

**Note:** This is a convenience function for users porting code from Matlab, and wraps *`standard_normal`*. That function takes a tuple to specify the size of the output, which is consistent with other NumPy functions like *`numpy.zeros`* and *`numpy.ones`*.

---

**Note:** New code should use the `standard_normal` method of a `default_rng()` instance instead; see *random-quick-start*.

---

If positive int_like arguments are provided, *`randn`* generates an array of shape `(d0, d1, ..., dn)`, filled with random floats sampled from a univariate "normal" (Gaussian) distribution of mean 0 and variance 1. A single float randomly sampled from the distribution is returned if no argument is provided.

**Parameters**

**d0, d1, …, dn**

[int, optional] The dimensions of the returned array, must be non-negative. If no argument is given a single Python float is returned.

**Returns**

**Z**

[ndarray or float] A `(d0, d1, ..., dn)`-shaped array of floating-point samples from the standard normal distribution, or a single such float if no parameters were supplied.

**See also:**

***`standard_normal`***

Similar, but takes a tuple as its argument.

***`normal`***

Also accepts mu and sigma arguments.

***`Generator.standard_normal`***

which should be used for new code.

### Notes

For random samples from $N(\mu, \sigma^2)$, use:

`sigma * np.random.randn(...) + mu`

### Examples

```
>>> np.random.randn()
2.1923875335537315  # random
```

Two-by-four array of samples from N(3, 6.25):

```
>>> 3 + 2.5 * np.random.randn(2, 4)
array([[-4.49401501,  4.00950034, -1.81814867,  7.29718677],   # random
       [ 0.39924804,  4.68456316,  4.99394529,  4.84057254]])  # random
```

`numpy.random.`**`random`**(*size=None*)

Return random floats in the half-open interval [0.0, 1.0). Alias for *`random_sample`* to ease forward-porting to the new random API.

`numpy.random.`**`random_integers`**(*low*, *high=None*, *size=None*)

Random integers of type *np.int_* between *low* and *high*, inclusive.

Return random integers of type *np.int_* from the "discrete uniform" distribution in the closed interval [*low*, *high*]. If *high* is None (the default), then results are from [1, *low*]. The *np.int_* type translates to the C long integer type and its precision is platform dependent.

This function has been deprecated. Use randint instead.

Deprecated since version 1.11.0.

**Parameters**

**low**

[int] Lowest (signed) integer to be drawn from the distribution (unless `high=None`, in which case this parameter is the *highest* such integer).

**high**

[int, optional] If provided, the largest (signed) integer to be drawn from the distribution (see above for behavior if `high=None`).

**size**

[int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. Default is None, in which case a single value is returned.

**Returns**

**out**

[int or ndarray of ints] *size*-shaped array of random integers from the appropriate distribution, or a single such random int if *size* not provided.

**See also:**

*randint*

Similar to *random_integers*, only for the half-open interval [*low*, *high*), and 0 is the lowest value if *high* is omitted.

### Notes

To sample from N evenly spaced floating-point numbers between a and b, use:

```
a + (b - a) * (np.random.random_integers(N) - 1) / (N - 1.)
```

### Examples

```
>>> np.random.random_integers(5)
4 # random
>>> type(np.random.random_integers(5))
<class 'numpy.int64'>
>>> np.random.random_integers(5, size=(3,2))
array([[5, 4], # random
       [3, 3],
       [4, 5]])
```

Choose five random numbers from the set of five evenly-spaced numbers between 0 and 2.5, inclusive (*i.e.*, from the set $0, 5/8, 10/8, 15/8, 20/8$):

```
>>> 2.5 * (np.random.random_integers(5, size=(5,)) - 1) / 4.
array([ 0.625,  1.25 ,  0.625,  0.625,  2.5  ]) # random
```

Roll two six sided dice 1000 times and sum the results:

```
>>> d1 = np.random.random_integers(1, 6, 1000)
>>> d2 = np.random.random_integers(1, 6, 1000)
>>> dsums = d1 + d2
```

Display results as a histogram:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(dsums, 11, density=True)
>>> plt.show()
```

`numpy.random.`**`random_sample`**`(`*size=None*`)`

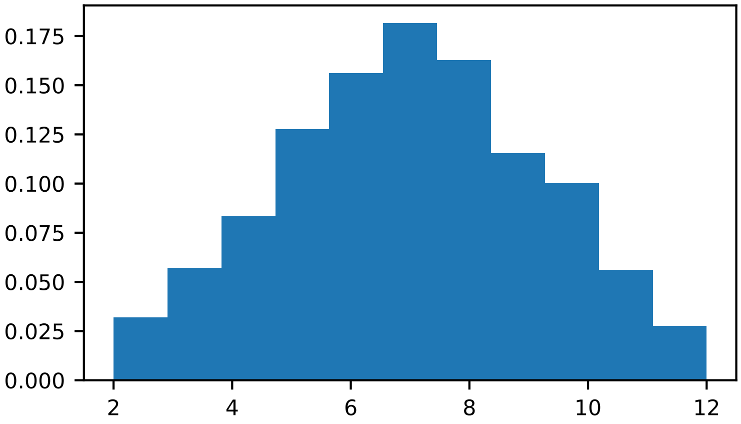
Return random floats in the half-open interval [0.0, 1.0).

Results are from the "continuous uniform" distribution over the stated interval. To sample $Unif[a, b), b > a$ multiply the output of *random_sample* by *(b-a)* and add *a*:

```
(b - a) * random_sample() + a
```

**Note:** New code should use the `random` method of a `default_rng()` instance instead; see *random-quick-start*.

**Parameters**



**size**

[int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. Default is None, in which case a single value is returned.

**Returns**

**out**

[float or ndarray of floats] Array of random floats of shape *size* (unless `size=None`, in which case a single float is returned).

**See also:**

***`Generator.random`***

which should be used for new code.

## Examples

```
>>> np.random.random_sample()
0.47108547995356098 # random
>>> type(np.random.random_sample())
<class 'float'>
>>> np.random.random_sample((5,))
array([ 0.30220482,  0.86820401,  0.1654503 ,  0.11659149,  0.54323428]) # random
```

Three-by-two array of random numbers from [-5, 0):

```
>>> 5 * np.random.random_sample((3, 2)) - 5
array([[-3.99149989, -0.52338984], # random
       [-2.99091858, -0.79479508],
       [-1.23204345, -1.75224494]])
```

`numpy.random.`**`ranf`**`()`

This is an alias of *`random_sample`*. See *`random_sample`* for the complete documentation.

`numpy.random.`**`rayleigh`**`(`*scale=1.0*, *size=None*`)`

Draw samples from a Rayleigh distribution.

The $\chi$ and Weibull distributions are generalizations of the Rayleigh.

---

**Note:** New code should use the `rayleigh` method of a `default_rng()` instance instead; see *random-quick-start*.

---

**Parameters**

**scale**

[float or array_like of floats, optional] Scale, also equals the mode. Must be non-negative. Default is 1.

**size**

[int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `scale` is a scalar. Otherwise, `np.array(scale).size` samples are drawn.

**Returns**

**out**

[ndarray or scalar] Drawn samples from the parameterized Rayleigh distribution.

**See also:**

***`Generator.rayleigh`***

which should be used for new code.

## Notes

The probability density function for the Rayleigh distribution is

$$P(x;scale) = \frac{x}{scale^2}e^{\frac{-x^2}{2 \cdot scale^2}}$$

The Rayleigh distribution would arise, for example, if the East and North components of the wind velocity had identical zero-mean Gaussian distributions. Then the wind speed would have a Rayleigh distribution.

## References

[1], [2]

## Examples

Draw values from the distribution and plot the histogram

```
>>> from matplotlib.pyplot import hist
>>> values = hist(np.random.rayleigh(3, 100000), bins=200, density=True)
```

Wave heights tend to follow a Rayleigh distribution. If the mean wave height is 1 meter, what fraction of waves are likely to be larger than 3 meters?

```
>>> meanvalue = 1
>>> modevalue = np.sqrt(2 / np.pi) * meanvalue
>>> s = np.random.rayleigh(modevalue, 1000000)
```

The percentage of waves larger than 3 meters is:

```
>>> 100.*sum(s>3)/1000000.
0.087300000000000003 # random
```

`numpy.random.`**`sample`**`()`

This is an alias of *`random_sample`*. See *`random_sample`* for the complete documentation.

`numpy.random.`**`seed`**`(`*self*, *seed=None*`)`

Reseed a legacy MT19937 BitGenerator

## Notes

This is a convenience, legacy function.

The best practice is to **not** reseed a BitGenerator, rather to recreate a new one. This method is here for legacy reasons. This example demonstrates best practice.

```
>>> from numpy.random import MT19937
>>> from numpy.random import RandomState, SeedSequence
>>> rs = RandomState(MT19937(SeedSequence(123456789)))
# Later, you want to restart the stream
>>> rs = RandomState(MT19937(SeedSequence(987654321)))
```

`numpy.random.`**`set_state`**`(`*state*`)`

Set the internal state of the generator from a tuple.

For use if one has reason to manually (re-)set the internal state of the bit generator used by the RandomState instance. By default, RandomState uses the "Mersenne Twister"[1] pseudo-random number generating algorithm.

**Parameters**

**state**

[{tuple(str, ndarray of 624 uints, int, int, float), dict}] The *state* tuple has the following items:

1. the string 'MT19937', specifying the Mersenne Twister algorithm.
2. a 1-D array of 624 unsigned integers `keys`.
3. an integer `pos`.
4. an integer `has_gauss`.
5. a float `cached_gaussian`.

If state is a dictionary, it is directly set using the BitGenerators *state* property.

**Returns**

**out**

[None] Returns 'None' on success.

**See also:**

*`get_state`*

### Notes

*`set_state`* and *`get_state`* are not needed to work with any of the random distributions in NumPy. If the internal state is manually altered, the user should know exactly what he/she is doing.

For backwards compatibility, the form (str, array of 624 uints, int) is also accepted although it is missing some information about the cached Gaussian value: `state = ('MT19937', keys, pos)`.

### References

[1]

`numpy.random.`**`shuffle`**`(`*x*`)`

Modify a sequence in-place by shuffling its contents.

This function only shuffles the array along the first axis of a multi-dimensional array. The order of sub-arrays is changed but their contents remains the same.

---

**Note:** New code should use the `shuffle` method of a `default_rng()` instance instead; see *random-quick-start*.

---

**Parameters**

**x**

[array_like] The array or list to be shuffled.

**Returns**

**None**

**See also:**

***`Generator.shuffle`***

which should be used for new code.

## Examples

```
>>> arr = np.arange(10)
>>> np.random.shuffle(arr)
>>> arr
[1 7 5 2 9 4 3 6 0 8] # random
```

Multi-dimensional arrays are only shuffled along the first axis:

```
>>> arr = np.arange(9).reshape((3, 3))
>>> np.random.shuffle(arr)
>>> arr
array([[3, 4, 5], # random
       [6, 7, 8],
       [0, 1, 2]])
```

`numpy.random.`**`standard_cauchy`**(*size=None*)

Draw samples from a standard Cauchy distribution with mode = 0.

Also known as the Lorentz distribution.

---

**Note:** New code should use the `standard_cauchy` method of a `default_rng()` instance instead; see *random-quick-start*.

---

**Parameters**

**size**

[int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. Default is None, in which case a single value is returned.

**Returns**

**samples**

[ndarray or scalar] The drawn samples.

**See also:**

***`Generator.standard_cauchy`***

which should be used for new code.

## Notes

The probability density function for the full Cauchy distribution is

$$P(x; x_0, \gamma) = \frac{1}{\pi\gamma\bigl[1 + (\frac{x-x_0}{\gamma})^2\bigr]}$$

and the Standard Cauchy distribution just sets $x_0 = 0$ and $\gamma = 1$

The Cauchy distribution arises in the solution to the driven harmonic oscillator problem, and also describes spectral line broadening. It also describes the distribution of values at which a line tilted at a random angle will cut the x axis.

When studying hypothesis tests that assume normality, seeing how the tests perform on data from a Cauchy distribution is a good indicator of their sensitivity to a heavy-tailed distribution, since the Cauchy looks very much like a Gaussian distribution, but with heavier tails.

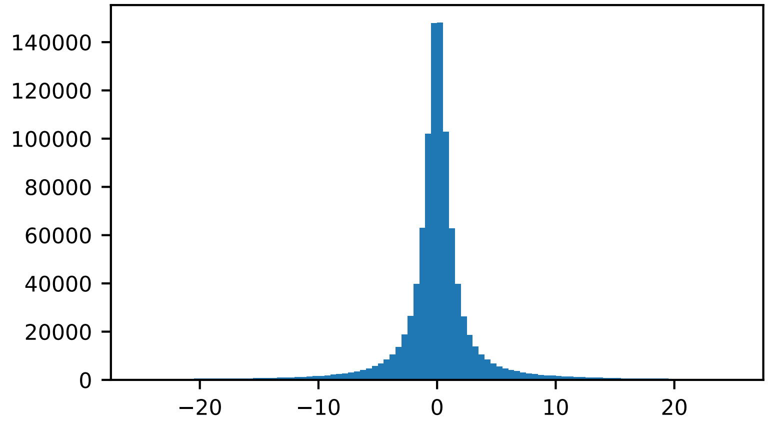
### References

[1], [2], [3]

### Examples

Draw samples and plot the distribution:

```
>>> import matplotlib.pyplot as plt
>>> s = np.random.standard_cauchy(1000000)
>>> s = s[(s>-25) & (s<25)]  # truncate distribution so it plots well
>>> plt.hist(s, bins=100)
>>> plt.show()
```



`numpy.random.`**`standard_exponential`**`(`*size=None*`)`

Draw samples from the standard exponential distribution.

*`standard_exponential`* is identical to the exponential distribution with a scale parameter of 1.

**Note:** New code should use the `standard_exponential` method of a `default_rng()` instance instead; see *random-quick-start*.

**Parameters**

**size**

[int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. Default is None, in which case a single value is returned.

**Returns**

**out**

[float or ndarray] Drawn samples.

**See also:**

*`Generator.standard_exponential`*

which should be used for new code.

## Examples

Output a 3x8000 array:

```
>>> n = np.random.standard_exponential((3, 8000))
```

`numpy.random.`**`standard_gamma`**(*shape*, *size=None*)

Draw samples from a standard Gamma distribution.

Samples are drawn from a Gamma distribution with specified parameters, shape (sometimes designated "k") and scale=1.

**Note:** New code should use the `standard_gamma` method of a `default_rng()` instance instead; see *random-quick-start*.

**Parameters**

**shape**

[float or array_like of floats] Parameter, must be non-negative.

**size**

[int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `shape` is a scalar. Otherwise, `np.array(shape).size` samples are drawn.

**Returns**

**out**

[ndarray or scalar] Drawn samples from the parameterized standard gamma distribution.

**See also:**

**`scipy.stats.gamma`**

probability density function, distribution or cumulative density function, etc.

*`Generator.standard_gamma`*

which should be used for new code.

### Notes

The probability density for the Gamma distribution is

$$p(x) = x^{k-1}\frac{e^{-x/\theta}}{\theta^k\Gamma(k)},$$

where $k$ is the shape and $\theta$ the scale, and $\Gamma$ is the Gamma function.

The Gamma distribution is often used to model the times to failure of electronic components, and arises naturally in processes for which the waiting times between Poisson distributed events are relevant.

### References

[1], [2]

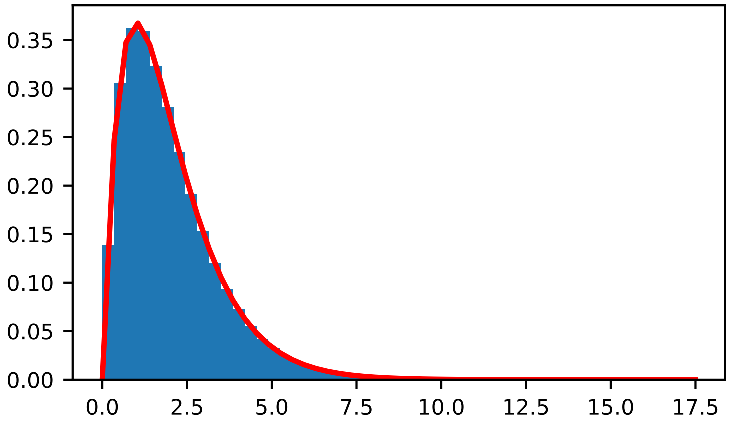
### Examples

Draw samples from the distribution:

```
>>> shape, scale = 2., 1. # mean and width
>>> s = np.random.standard_gamma(shape, 1000000)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> import scipy.special as sps
>>> count, bins, ignored = plt.hist(s, 50, density=True)
>>> y = bins**(shape-1) * ((np.exp(-bins/scale))/
...                       (sps.gamma(shape) * scale**shape))
>>> plt.plot(bins, y, linewidth=2, color='r')
>>> plt.show()
```



`numpy.random.`**`standard_normal`**(*size=None*)

Draw samples from a standard Normal distribution (mean=0, stdev=1).

---

**Note:** New code should use the `standard_normal` method of a `default_rng()` instance instead; see *random-quick-start*.

---

**Parameters**

**size**

[int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. Default is None, in which case a single value is returned.

**Returns**

**out**

[float or ndarray] A floating-point array of shape `size` of drawn samples, or a single sample if `size` was not specified.

**See also:**

***normal***

Equivalent function with additional `loc` and `scale` arguments for setting the mean and standard deviation.

***Generator.standard_normal***

which should be used for new code.

## Notes

For random samples from $N(\mu, \sigma^2)$, use one of:

```
mu + sigma * np.random.standard_normal(size=...)
np.random.normal(mu, sigma, size=...)
```

## Examples

```
>>> np.random.standard_normal()
2.1923875335537315 #random
```

```
>>> s = np.random.standard_normal(8000)
>>> s
array([ 0.6888893 ,  0.78096262, -0.89086505, ...,  0.49876311,  # random
       -0.38672696, -0.4685006 ])                                # random
>>> s.shape
(8000,)
>>> s = np.random.standard_normal(size=(3, 4, 2))
>>> s.shape
(3, 4, 2)
```

Two-by-four array of samples from $N(3, 6.25)$:

```
>>> 3 + 2.5 * np.random.standard_normal(size=(2, 4))
array([[-4.49401501,  4.00950034, -1.81814867,  7.29718677],   # random
       [ 0.39924804,  4.68456316,  4.99394529,  4.84057254]])  # random
```

`numpy.random.`**`standard_t`**(*df*, *size=None*)

Draw samples from a standard Student's t distribution with *df* degrees of freedom.

A special case of the hyperbolic distribution. As *df* gets large, the result resembles that of the standard normal distribution (*`standard_normal`*).

**Note:** New code should use the `standard_t` method of a `default_rng()` instance instead; see *random-quick-start*.

**Parameters**

**df**

[float or array_like of floats] Degrees of freedom, must be > 0.

**size**

[int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `df` is a scalar. Otherwise, `np.array(df).size` samples are drawn.

**Returns**

**out**

[ndarray or scalar] Drawn samples from the parameterized standard Student's t distribution.

**See also:**

***`Generator.standard_t`***

which should be used for new code.

## Notes

The probability density function for the t distribution is

$$P(x, df) = \frac{\Gamma(\frac{df+1}{2})}{\sqrt{\pi df}\Gamma(\frac{df}{2})}\Bigl( 1+\frac{x^2}{df} \Bigr)^{-(df+1)/2}$$

The t test is based on an assumption that the data come from a Normal distribution. The t test provides a way to test whether the sample mean (that is the mean calculated from the data) is a good estimate of the true mean.

The derivation of the t-distribution was first published in 1908 by William Gosset while working for the Guinness Brewery in Dublin. Due to proprietary issues, he had to publish under a pseudonym, and so he used the name Student.

### References

[1], [2]

### Examples

From Dalgaard page 83 [1], suppose the daily energy intake for 11 women in kilojoules (kJ) is:

```
>>> intake = np.array([5260., 5470, 5640, 6180, 6390, 6515, 6805, 7515, \
...                    7515, 8230, 8770])
```

Does their energy intake deviate systematically from the recommended value of 7725 kJ?

We have 10 degrees of freedom, so is the sample mean within 95% of the recommended value?

```
>>> s = np.random.standard_t(10, size=100000)
>>> np.mean(intake)
6753.636363636364
>>> intake.std(ddof=1)
1142.1232221373727
```

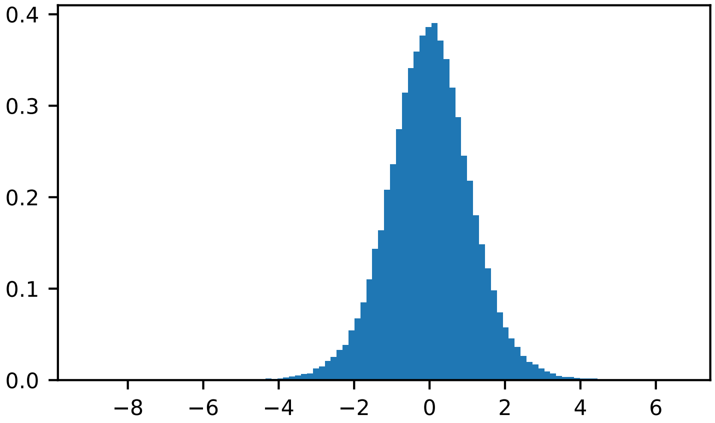
Calculate the t statistic, setting the ddof parameter to the unbiased value so the divisor in the standard deviation will be degrees of freedom, N-1.

```
>>> t = (np.mean(intake)-7725)/(intake.std(ddof=1)/np.sqrt(len(intake)))
>>> import matplotlib.pyplot as plt
>>> h = plt.hist(s, bins=100, density=True)
```

For a one-sided t-test, how far out in the distribution does the t statistic appear?

```
>>> np.sum(s<t) / float(len(s))
0.0090699999999999999  #random
```

So the p-value is about 0.009, which says the null hypothesis has a probability of about 99% of being true.



`numpy.random.`**`triangular`**(*left*, *mode*, *right*, *size=None*)

Draw samples from the triangular distribution over the interval `[left, right]`.

The triangular distribution is a continuous probability distribution with lower limit left, peak at mode, and upper limit right. Unlike the other distributions, these parameters directly define the shape of the pdf.

---

**Note:** New code should use the `triangular` method of a `default_rng()` instance instead; see *random-quick-start*.

---

**Parameters**

**left**

[float or array_like of floats] Lower limit.

**mode**

[float or array_like of floats] The value where the peak of the distribution occurs. The value must fulfill the condition `left <= mode <= right`.

**right**

[float or array_like of floats] Upper limit, must be larger than *left*.

**size**

[int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `left`, `mode`, and `right` are all scalars. Otherwise, `np.broadcast(left, mode, right).size` samples are drawn.

**Returns**

**out**

[ndarray or scalar] Drawn samples from the parameterized triangular distribution.

**See also:**

**`Generator.triangular`**

which should be used for new code.

## Notes

The probability density function for the triangular distribution is

$$P(x;l,m,r) = \begin{cases} \frac{2(x-l)}{(r-l)(m-l)} & \text{for } l \leq x \leq m, \\ \frac{2(r-x)}{(r-l)(r-m)} & \text{for } m \leq x \leq r, \\ 0 & \text{otherwise.} \end{cases}$$

The triangular distribution is often used in ill-defined problems where the underlying distribution is not known, but some knowledge of the limits and mode exists. Often it is used in simulations.

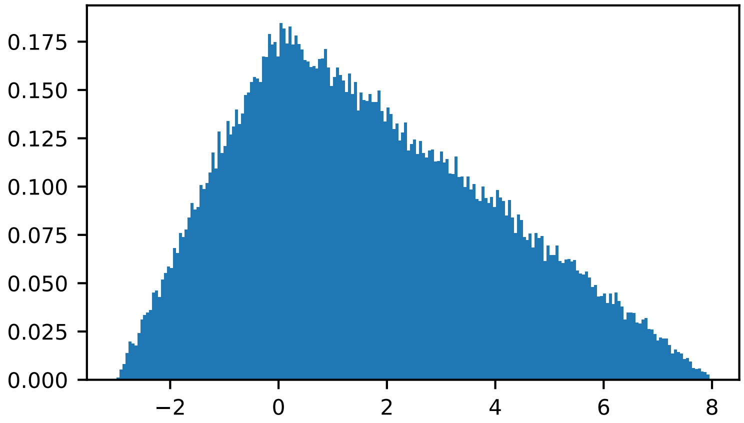
### References

[1]

### Examples

Draw values from the distribution and plot the histogram:

```
>>> import matplotlib.pyplot as plt
>>> h = plt.hist(np.random.triangular(-3, 0, 8, 100000), bins=200,
...              density=True)
>>> plt.show()
```



`numpy.random.`**`uniform`**(*low=0.0, high=1.0, size=None*)

Draw samples from a uniform distribution.

Samples are uniformly distributed over the half-open interval `[low, high)` (includes low, but excludes high). In other words, any value within the given interval is equally likely to be drawn by *`uniform`*.

**Note:** New code should use the `uniform` method of a `default_rng()` instance instead; see *random-quick-start*.

**Parameters**

**low**

[float or array_like of floats, optional] Lower boundary of the output interval. All values generated will be greater than or equal to low. The default value is 0.

**high**

[float or array_like of floats] Upper boundary of the output interval. All values generated will be less than or equal to high. The default value is 1.0.

**size**

[int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `low` and `high` are both scalars. Otherwise, `np.broadcast(low, high).size` samples are drawn.

**Returns**

**out**

[ndarray or scalar] Drawn samples from the parameterized uniform distribution.

**See also:**

***`randint`***

Discrete uniform distribution, yielding integers.

***`random_integers`***

Discrete uniform distribution over the closed interval `[low, high]`.

***`random_sample`***

Floats uniformly distributed over `[0, 1)`.

***`random`***

Alias for *`random_sample`*.

***`rand`***

Convenience function that accepts dimensions as input, e.g., `rand(2,2)` would generate a 2-by-2 array of floats, uniformly distributed over `[0, 1)`.

***`Generator.uniform`***

which should be used for new code.

## Notes

The probability density function of the uniform distribution is

$$p(x) = \frac{1}{b - a}$$

anywhere within the interval `[a, b)`, and zero elsewhere.

When `high` == `low`, values of `low` will be returned. If `high` < `low`, the results are officially undefined and may eventually raise an error, i.e. do not rely on this function to behave when passed arguments satisfying that inequality condition. The `high` limit may be included in the returned array of floats due to floating-point rounding in the equation `low + (high-low) * random_sample()`. For example:

```
>>> x = np.float32(5*0.99999999)
>>> x
5.0
```

## Examples

Draw samples from the distribution:

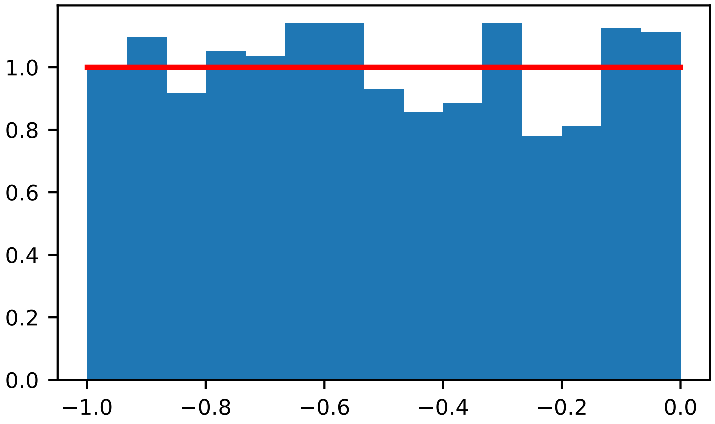
```
>>> s = np.random.uniform(-1,0,1000)
```

All values are within the given interval:

```
>>> np.all(s >= -1)
True
>>> np.all(s < 0)
True
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 15, density=True)
>>> plt.plot(bins, np.ones_like(bins), linewidth=2, color='r')
>>> plt.show()
```



`numpy.random.`**`vonmises`**`(`*mu*, *kappa*, *size=None*`)`

Draw samples from a von Mises distribution.

Samples are drawn from a von Mises distribution with specified mode (mu) and dispersion (kappa), on the interval [-pi, pi].

The von Mises distribution (also known as the circular normal distribution) is a continuous probability distribution on the unit circle. It may be thought of as the circular analogue of the normal distribution.

**Note:** New code should use the `vonmises` method of a `default_rng()` instance instead; see *random-quick-start*.

**Parameters**

**mu**

[float or array_like of floats] Mode (“center”) of the distribution.

**kappa**

[float or array_like of floats] Dispersion of the distribution, has to be >=0.

**size**

[int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `mu` and `kappa` are both scalars. Otherwise, `np.broadcast(mu, kappa).size` samples are drawn.

**Returns**

**out**

[ndarray or scalar] Drawn samples from the parameterized von Mises distribution.

**See also:**

**`scipy.stats.vonmises`**

probability density function, distribution, or cumulative density function, etc.

***`Generator.vonmises`***

which should be used for new code.

## Notes

The probability density for the von Mises distribution is

$$p(x) = \frac{e^{\kappa cos(x-\mu)}}{2\pi I_0(\kappa)},$$

where $\mu$ is the mode and $\kappa$ the dispersion, and $I_0(\kappa)$ is the modified Bessel function of order 0.

The von Mises is named for Richard Edler von Mises, who was born in Austria-Hungary, in what is now the Ukraine. He fled to the United States in 1939 and became a professor at Harvard. He worked in probability theory, aerodynamics, fluid mechanics, and philosophy of science.

## References

[1], [2]

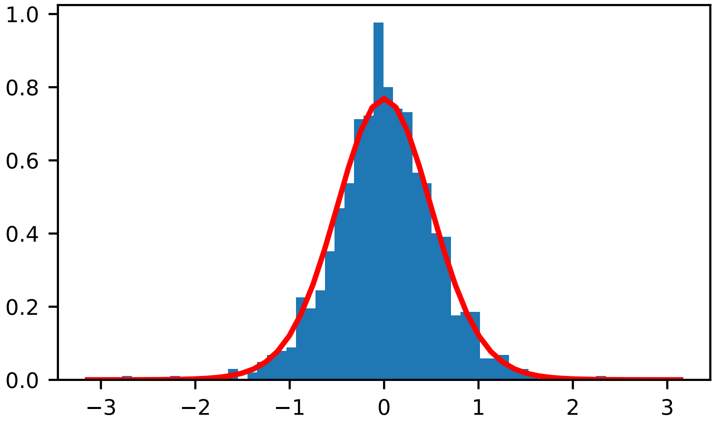
## Examples

Draw samples from the distribution:

```
>>> mu, kappa = 0.0, 4.0 # mean and dispersion
>>> s = np.random.vonmises(mu, kappa, 1000)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> from scipy.special import i0
>>> plt.hist(s, 50, density=True)
>>> x = np.linspace(-np.pi, np.pi, num=51)
>>> y = np.exp(kappa*np.cos(x-mu))/(2*np.pi*i0(kappa))
>>> plt.plot(x, y, linewidth=2, color='r')
>>> plt.show()
```



`numpy.random.`**`wald`**`(`*mean*, *scale*, *size=None*`)`

Draw samples from a Wald, or inverse Gaussian, distribution.

As the scale approaches infinity, the distribution becomes more like a Gaussian. Some references claim that the Wald is an inverse Gaussian with mean equal to 1, but this is by no means universal.

The inverse Gaussian distribution was first studied in relationship to Brownian motion. In 1956 M.C.K. Tweedie used the name inverse Gaussian because there is an inverse relationship between the time to cover a unit distance and distance covered in unit time.

**Note:** New code should use the `wald` method of a `default_rng()` instance instead; see *random-quick-start*.

**Parameters**

**mean**

[float or array_like of floats] Distribution mean, must be > 0.

**scale**

[float or array_like of floats] Scale parameter, must be > 0.

**size**

[int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `mean` and `scale` are both scalars. Otherwise, `np.broadcast(mean, scale).size` samples are drawn.

**Returns**

**out**

[ndarray or scalar] Drawn samples from the parameterized Wald distribution.

**See also:**

*`Generator.wald`*

which should be used for new code.

## Notes

The probability density function for the Wald distribution is

$$P(x;mean,scale) = \sqrt{\frac{scale}{2\pi x^3}}e^{\frac{-scale(x-mean)^2}{2\cdot mean^2 x}}$$

As noted above the inverse Gaussian distribution first arise from attempts to model Brownian motion. It is also a competitor to the Weibull for use in reliability modeling and modeling stock returns and interest rate processes.

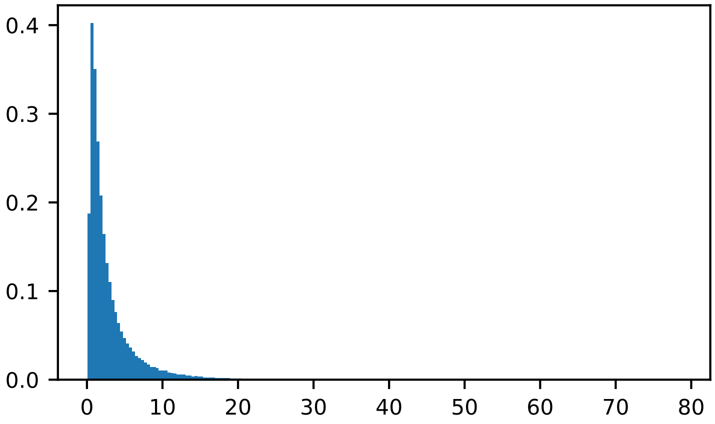
## References

[1], [2], [3]

## Examples

Draw values from the distribution and plot the histogram:

```
>>> import matplotlib.pyplot as plt
>>> h = plt.hist(np.random.wald(3, 2, 100000), bins=200, density=True)
>>> plt.show()
```



`numpy.random.weibull(a, size=None)`

Draw samples from a Weibull distribution.

Draw samples from a 1-parameter Weibull distribution with the given shape parameter *a*.

$$X = (-ln(U))^{1/a}$$

Here, U is drawn from the uniform distribution over (0,1].

The more common 2-parameter Weibull, including a scale parameter $\lambda$ is just $X = \lambda(-ln(U))^{1/a}$.

---

**Note:** New code should use the `weibull` method of a `default_rng()` instance instead; see *random-quick-start*.

---

**Parameters**

**a**

[float or array_like of floats] Shape parameter of the distribution. Must be nonnegative.

**size**

[int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `a` is a scalar. Otherwise, `np.array(a).size` samples are drawn.

**Returns**

**out**

[ndarray or scalar] Drawn samples from the parameterized Weibull distribution.

**See also:**

`scipy.stats.weibull_max`, `scipy.stats.weibull_min`, `scipy.stats.genextreme`, *`gumbel`*

***`Generator.weibull`***

which should be used for new code.

## Notes

The Weibull (or Type III asymptotic extreme value distribution for smallest values, SEV Type III, or Rosin-Rammler distribution) is one of a class of Generalized Extreme Value (GEV) distributions used in modeling extreme value problems. This class includes the Gumbel and Frechet distributions.

The probability density for the Weibull distribution is

$$p(x) = \frac{a}{\lambda}(\frac{x}{\lambda})^{a-1}e^{-(x/\lambda)^a},$$

where $a$ is the shape and $\lambda$ the scale.

The function has its peak (the mode) at $\lambda(\frac{a-1}{a})^{1/a}$.

When `a = 1`, the Weibull distribution reduces to the exponential distribution.

### References

[1], [2], [3]

### Examples

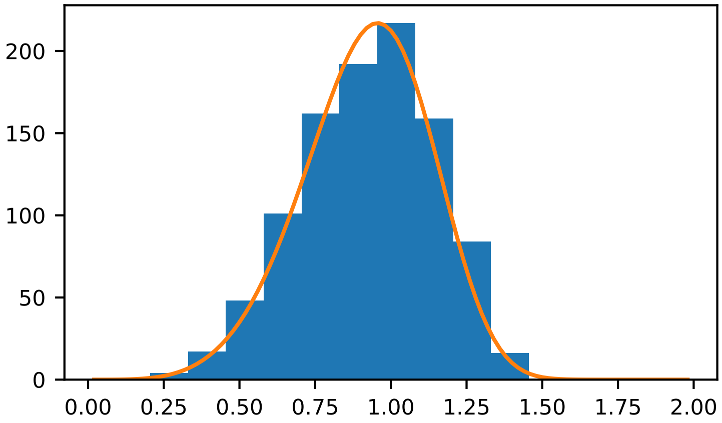
Draw samples from the distribution:

```
>>> a = 5. # shape
>>> s = np.random.weibull(a, 1000)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> x = np.arange(1,100.)/50.
>>> def weib(x,n,a):
...     return (a / n) * (x / n)**(a - 1) * np.exp(-(x / n)**a)
```

```
>>> count, bins, ignored = plt.hist(np.random.weibull(5.,1000))
>>> x = np.arange(1,100.)/50.
>>> scale = count.max()/weib(x, 1., 5.).max()
>>> plt.plot(x, weib(x, 1., 5.)*scale)
>>> plt.show()
```



`numpy.random.`**`zipf`**(*a*, *size=None*)

Draw samples from a Zipf distribution.

Samples are drawn from a Zipf distribution with specified parameter $a > 1$.

The Zipf distribution (also known as the zeta distribution) is a continuous probability distribution that satisfies Zipf's law: the frequency of an item is inversely proportional to its rank in a frequency table.

**Note:** New code should use the `zipf` method of a `default_rng()` instance instead; see *random-quick-start*.

**Parameters**

**a**

[float or array_like of floats] Distribution parameter. Must be greater than 1.

**size**

[int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `a` is a scalar. Otherwise, `np.array(a).size` samples are drawn.

**Returns**

**out**

[ndarray or scalar] Drawn samples from the parameterized Zipf distribution.

**See also:**

**`scipy.stats.zipf`**

probability density function, distribution, or cumulative density function, etc.

***`Generator.zipf`***

which should be used for new code.

## Notes

The probability density for the Zipf distribution is

$$p(x) = \frac{x^{-a}}{\zeta(a)},$$

where $\zeta$ is the Riemann Zeta function.

It is named for the American linguist George Kingsley Zipf, who noted that the frequency of any word in a sample of a language is inversely proportional to its rank in the frequency table.

## References

[1]

## Examples

Draw samples from the distribution:

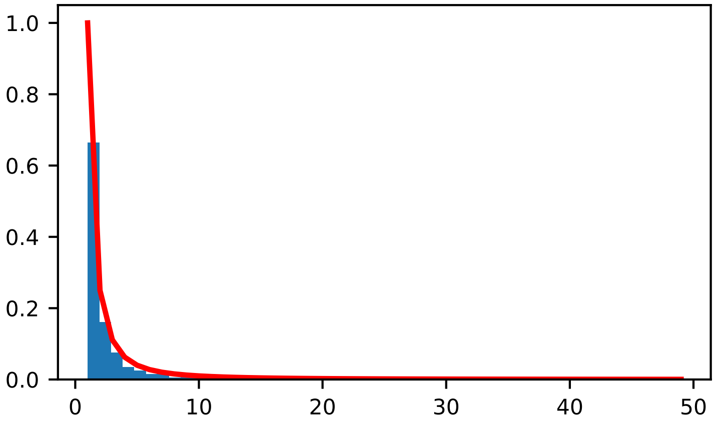
```
>>> a = 2. # parameter
>>> s = np.random.zipf(a, 1000)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> from scipy import special
```

Truncate s values at 50 so plot is interesting:

```
>>> count, bins, ignored = plt.hist(s[s<50], 50, density=True)
>>> x = np.arange(1., 50.)
>>> y = x**(-a) / special.zetac(a)
>>> plt.plot(x, y/max(y), linewidth=2, color='r')
>>> plt.show()
```



#### Bit Generators

The random values produced by `Generator` orignate in a BitGenerator. The BitGenerators do not directly provide random numbers and only contains methods used for seeding, getting or setting the state, jumping or advancing the state, and for accessing low-level wrappers for consumption by code that can efficiently access the functions provided, e.g., numba.

#### Supported BitGenerators

The included BitGenerators are:

- PCG-64 - The default. A fast generator that supports many parallel streams and can be advanced by an arbitrary amount. See the documentation for `advance`. PCG-64 has a period of $2^{128}$. See the PCG author's page for more details about this class of PRNG.
- MT19937 - The standard Python BitGenerator. Adds a `MT19937.jumped` function that returns a new generator with state as-if $2^{128}$ draws have been made.
- Philox - A counter-based generator capable of being advanced an arbitrary number of steps or generating independent streams. See the Random123 page for more details about this class of bit generators.
- SFC64 - A fast generator based on random invertible mappings. Usually the fastest generator of the four. See the SFC author's page for (a little) more detail.

| | |
|---|---|
| `BitGenerator`([seed]) | Base Class for generic BitGenerators, which provide a stream of random bits based on different algorithms. |

**class** `numpy.random.`**`BitGenerator`**(*seed=None*)

Base Class for generic BitGenerators, which provide a stream of random bits based on different algorithms. Must be overridden.

**Parameters**

**seed**

[{None, int, array_like[ints], SeedSequence}, optional] A seed to initialize the *`BitGenerator`*. If None, then fresh, unpredictable entropy will be pulled from the OS. If an `int` or `array_like[ints]` is passed, then it will be passed to ~‘numpy.random.SeedSequence‘ to derive the initial *`BitGenerator`* state. One may also pass in a *`SeedSequence`* instance.

**Attributes**

**lock**

[threading.Lock] Lock instance that is shared so that the same BitGenerator can be used in multiple Generators without corrupting the state. Code that generates values from a bit generator should hold the bit generator’s lock.

**See Also**

**——-**

**SeedSequence**

## Methods

| | |
|---|---|
| *`random_raw`*(self[, size]) | Return randoms as generated by the underlying BitGenerator |

method

`BitGenerator.`**`random_raw`**(*self*, *size=None*)

Return randoms as generated by the underlying BitGenerator

**Parameters**

**size**

[int or tuple of ints, optional] Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. Default is None, in which case a single value is returned.

**output**

[bool, optional] Output values. Used for performance testing since the generated values are not returned.

**Returns**

**out**

[uint or ndarray] Drawn samples.

### Notes

This method directly exposes the the raw underlying pseudo-random number generator. All values are returned as unsigned 64-bit values irrespective of the number of bits produced by the PRNG.

See the class docstring for the number of bits returned.

## Mersenne Twister (MT19937)

**class** `numpy.random.`**`MT19937`**`(`*seed=None*`)`

Container for the Mersenne Twister pseudo-random number generator.

**Parameters**

**seed**

[{None, int, array_like[ints], SeedSequence}, optional] A seed to initialize the *`BitGenerator`*. If None, then fresh, unpredictable entropy will be pulled from the OS. If an `int` or `array_like[ints]` is passed, then it will be passed to *`SeedSequence`* to derive the initial *`BitGenerator`* state. One may also pass in a *`SeedSequence`* instance.

### Notes

`MT19937` provides a capsule containing function pointers that produce doubles, and unsigned 32 and 64- bit integers [1]. These are not directly consumable in Python and must be consumed by a `Generator` or similar object that supports low-level access.

The Python stdlib module "random" also contains a Mersenne Twister pseudo-random number generator.

**State and Seeding**

The `MT19937` state vector consists of a 624-element array of 32-bit unsigned integers plus a single integer value between 0 and 624 that indexes the current position within the main array.

The input seed is processed by *`SeedSequence`* to fill the whole state. The first element is reset such that only its most significant bit is set.

**Parallel Features**

The preferred way to use a BitGenerator in parallel applications is to use the *`SeedSequence.spawn`* method to obtain entropy values, and to use these to generate new BitGenerators:

```
>>> from numpy.random import Generator, MT19937, SeedSequence
>>> sg = SeedSequence(1234)
>>> rg = [Generator(MT19937(s)) for s in sg.spawn(10)]
```

Another method is to use *`MT19937.jumped`* which advances the state as-if $2^{128}$ random numbers have been generated ([1], [2]). This allows the original sequence to be split so that distinct segments can be used in each worker process. All generators should be chained to ensure that the segments come from the same sequence.

```
>>> from numpy.random import Generator, MT19937, SeedSequence
>>> sg = SeedSequence(1234)
>>> bit_generator = MT19937(sg)
>>> rg = []
>>> for _ in range(10):
...     rg.append(Generator(bit_generator))
...     # Chain the BitGenerators
...     bit_generator = bit_generator.jumped()
```

**Compatibility Guarantee**

`MT19937` makes a guarantee that a fixed seed and will always produce the same random integer stream.

### References

[1], [2]

**Attributes**

**lock: threading.Lock**

Lock instance that is shared so that the same bit git generator can be used in multiple Generators without corrupting the state. Code that generates values from a bit generator should hold the bit generator's lock.

## State

| | |
|---|---|
| `state` | Get or set the PRNG state |

attribute

`MT19937.`**`state`**

Get or set the PRNG state

**Returns**

**state**

[dict] Dictionary containing the information required to describe the state of the PRNG

## Parallel generation

| | |
|---|---|
| `jumped`([jumps]) | Returns a new bit generator with the state jumped |

method

`MT19937.`**`jumped`**`(`*jumps=1*`)`

Returns a new bit generator with the state jumped

The state of the returned big generator is jumped as-if 2**(128 * jumps) random numbers have been generated.

**Parameters**

**jumps**

[integer, positive] Number of times to jump the state of the bit generator returned

**Returns**

**bit_generator**

[MT19937] New instance of generator jumped iter times

### Notes

The jump step is computed using a modified version of Matsumoto's implementation of Horner's method. The step polynomial is precomputed to perform 2**128 steps. The jumped state has been verified to match the state produced using Matsumoto's original code.

### References

[1], [2]

## Extending

| | |
|---|---|
| *cffi* | CFFI interface |
| *ctypes* | ctypes interface |

attribute

`MT19937.`**`cffi`**

CFFI interface

**Returns**

**interface**

[namedtuple] Named tuple containing CFFI wrapper

- state_address - Memory address of the state struct
- state - pointer to the state struct
- next_uint64 - function pointer to produce 64 bit integers
- next_uint32 - function pointer to produce 32 bit integers
- next_double - function pointer to produce doubles
- bitgen - pointer to the bit generator struct

attribute

`MT19937.`**`ctypes`**

ctypes interface

**Returns**

**interface**

[namedtuple] Named tuple containing ctypes wrapper

- state_address - Memory address of the state struct
- state - pointer to the state struct
- next_uint64 - function pointer to produce 64 bit integers
- next_uint32 - function pointer to produce 32 bit integers
- next_double - function pointer to produce doubles
- bitgen - pointer to the bit generator struct

### Permuted Congruential Generator (64-bit, PCG64)

**class** `numpy.random.`**`PCG64`**(*seed=None*)

BitGenerator for the PCG-64 pseudo-random number generator.

**Parameters**

**seed**

[{None, int, array_like[ints], SeedSequence}, optional] A seed to initialize the *`BitGenerator`*. If None, then fresh, unpredictable entropy will be pulled from the OS. If an `int` or `array_like[ints]` is passed, then it will be passed to *`SeedSequence`* to derive the initial *`BitGenerator`* state. One may also pass in a *`SeedSequence`* instance.

#### Notes

PCG-64 is a 128-bit implementation of O'Neill's permutation congruential generator ([1], [2]). PCG-64 has a period of $2^{128}$ and supports advancing an arbitrary number of steps as well as $2^{127}$ streams. The specific member of the PCG family that we use is PCG XSL RR 128/64 as described in the paper ([2]).

`PCG64` provides a capsule containing function pointers that produce doubles, and unsigned 32 and 64- bit integers. These are not directly consumable in Python and must be consumed by a `Generator` or similar object that supports low-level access.

Supports the method *`advance`* to advance the RNG an arbitrary number of steps. The state of the PCG-64 RNG is represented by 2 128-bit unsigned integers.

**State and Seeding**

The `PCG64` state vector consists of 2 unsigned 128-bit values, which are represented externally as Python ints. One is the state of the PRNG, which is advanced by a linear congruential generator (LCG). The second is a fixed odd increment used in the LCG.

The input seed is processed by *`SeedSequence`* to generate both values. The increment is not independently settable.

**Parallel Features**

The preferred way to use a BitGenerator in parallel applications is to use the *`SeedSequence.spawn`* method to obtain entropy values, and to use these to generate new BitGenerators:

```
>>> from numpy.random import Generator, PCG64, SeedSequence
>>> sg = SeedSequence(1234)
>>> rg = [Generator(PCG64(s)) for s in sg.spawn(10)]
```

**Compatibility Guarantee**

`PCG64` makes a guarantee that a fixed seed and will always produce the same random integer stream.

### References

[1], [2]

## State

| | |
|---|---|
| *state* | Get or set the PRNG state |

attribute

`PCG64.`**`state`**

Get or set the PRNG state

**Returns**

**state**

[dict] Dictionary containing the information required to describe the state of the PRNG

## Parallel generation

| | |
|---|---|
| *advance*(delta) | Advance the underlying RNG as-if delta draws have occurred. |
| *jumped*([jumps]) | Returns a new bit generator with the state jumped. |

method

`PCG64.`**`advance`**(*delta*)

Advance the underlying RNG as-if delta draws have occurred.

**Parameters**

**delta**

[integer, positive] Number of draws to advance the RNG. Must be less than the size state variable in the underlying RNG.

**Returns**

**self**

[PCG64] RNG advanced delta steps

### Notes

Advancing a RNG updates the underlying RNG state as-if a given number of calls to the underlying RNG have been made. In general there is not a one-to-one relationship between the number output random values from a particular distribution and the number of draws from the core RNG. This occurs for two reasons:

- The random values are simulated using a rejection-based method and so, on average, more than one value from the underlying RNG is required to generate an single draw.
- The number of bits required to generate a simulated value differs from the number of bits generated by the underlying RNG. For example, two 16-bit integer values can be simulated from a single draw of a 32-bit RNG.

Advancing the RNG state resets any pre-computed random numbers. This is required to ensure exact reproducibility.

method

`PCG64.`**`jumped`**`(`*jumps=1*`)`

Returns a new bit generator with the state jumped.

Jumps the state as-if jumps * 210306068529402873165736369884012333109 random numbers have been generated.

**Parameters**

**jumps**

[integer, positive] Number of times to jump the state of the bit generator returned

**Returns**

**bit_generator**

[PCG64] New instance of generator jumped iter times

### Notes

The step size is phi-1 when multiplied by 2**128 where phi is the golden ratio.

## Extending

| | |
|---|---|
| *cffi* | CFFI interface |
| *ctypes* | ctypes interface |

attribute

`PCG64.`**`cffi`**

CFFI interface

**Returns**

**interface**

[namedtuple] Named tuple containing CFFI wrapper

- state_address - Memory address of the state struct

- state - pointer to the state struct
- next_uint64 - function pointer to produce 64 bit integers
- next_uint32 - function pointer to produce 32 bit integers
- next_double - function pointer to produce doubles
- bitgen - pointer to the bit generator struct

attribute

`PCG64.`**`ctypes`**

ctypes interface

**Returns**

**interface**

[namedtuple] Named tuple containing ctypes wrapper

- state_address - Memory address of the state struct
- state - pointer to the state struct
- next_uint64 - function pointer to produce 64 bit integers
- next_uint32 - function pointer to produce 32 bit integers
- next_double - function pointer to produce doubles
- bitgen - pointer to the bit generator struct

### Philox Counter-based RNG

**`class`** `numpy.random.`**`Philox`**(*seed=None*, *counter=None*, *key=None*)

Container for the Philox (4x64) pseudo-random number generator.

**Parameters**

**seed**

[{None, int, array_like[ints], SeedSequence}, optional] A seed to initialize the *`BitGenerator`*. If None, then fresh, unpredictable entropy will be pulled from the OS. If an `int` or `array_like[ints]` is passed, then it will be passed to *`SeedSequence`* to derive the initial *`BitGenerator`* state. One may also pass in a *`SeedSequence`* instance.

**counter**

[{None, int, array_like}, optional] Counter to use in the Philox state. Can be either a Python int (long in 2.x) in [0, 2**256) or a 4-element uint64 array. If not provided, the RNG is initialized at 0.

**key**

[{None, int, array_like}, optional] Key to use in the Philox state. Unlike `seed`, the value in key is directly set. Can be either a Python int in [0, 2**128) or a 2-element uint64 array. *key* and `seed` cannot both be used.

## Notes

Philox is a 64-bit PRNG that uses a counter-based design based on weaker (and faster) versions of cryptographic functions [1]. Instances using different values of the key produce independent sequences. Philox has a period of $2^{256} - 1$ and supports arbitrary advancing and jumping the sequence in increments of $2^{128}$. These features allow multiple non-overlapping sequences to be generated.

`Philox` provides a capsule containing function pointers that produce doubles, and unsigned 32 and 64- bit integers. These are not directly consumable in Python and must be consumed by a `Generator` or similar object that supports low-level access.

**State and Seeding**

The `Philox` state vector consists of a 256-bit value encoded as a 4-element uint64 array and a 128-bit value encoded as a 2-element uint64 array. The former is a counter which is incremented by 1 for every 4 64-bit randoms produced. The second is a key which determined the sequence produced. Using different keys produces independent sequences.

The input `seed` is processed by `SeedSequence` to generate the key. The counter is set to 0.

Alternately, one can omit the `seed` parameter and set the `key` and `counter` directly.

**Parallel Features**

The preferred way to use a BitGenerator in parallel applications is to use the `SeedSequence.spawn` method to obtain entropy values, and to use these to generate new BitGenerators:

```
>>> from numpy.random import Generator, Philox, SeedSequence
>>> sg = SeedSequence(1234)
>>> rg = [Generator(Philox(s)) for s in sg.spawn(10)]
```

`Philox` can be used in parallel applications by calling the `jumped` method to advances the state as-if $2^{128}$ random numbers have been generated. Alternatively, `advance` can be used to advance the counter for any positive step in [0, 2**256). When using `jumped`, all generators should be chained to ensure that the segments come from the same sequence.

```
>>> from numpy.random import Generator, Philox
>>> bit_generator = Philox(1234)
>>> rg = []
>>> for _ in range(10):
...     rg.append(Generator(bit_generator))
...     bit_generator = bit_generator.jumped()
```

Alternatively, `Philox` can be used in parallel applications by using a sequence of distinct keys where each instance uses different key.

```
>>> key = 2**96 + 2**33 + 2**17 + 2**9
>>> rg = [Generator(Philox(key=key+i)) for i in range(10)]
```

**Compatibility Guarantee**

`Philox` makes a guarantee that a fixed `seed` will always produce the same random integer stream.

### References

[1]

### Examples

```
>>> from numpy.random import Generator, Philox
>>> rg = Generator(Philox(1234))
>>> rg.standard_normal()
0.123  # random
```

**Attributes**

**lock: threading.Lock**

Lock instance that is shared so that the same bit git generator can be used in multiple Generators without corrupting the state. Code that generates values from a bit generator should hold the bit generator's lock.

## State

| | |
|---|---|
| *`state`* | Get or set the PRNG state |

attribute

`Philox.`**`state`**

Get or set the PRNG state

**Returns**

**state**

[dict] Dictionary containing the information required to describe the state of the PRNG

## Parallel generation

| | |
|---|---|
| *`advance`*(delta) | Advance the underlying RNG as-if delta draws have occurred. |
| *`jumped`*([jumps]) | Returns a new bit generator with the state jumped |

method

`Philox.`**`advance`**(*delta*)

Advance the underlying RNG as-if delta draws have occurred.

**Parameters**

**delta**

[integer, positive] Number of draws to advance the RNG. Must be less than the size state variable in the underlying RNG.

**Returns**

**self**

[Philox] RNG advanced delta steps

### Notes

Advancing a RNG updates the underlying RNG state as-if a given number of calls to the underlying RNG have been made. In general there is not a one-to-one relationship between the number output random values from a particular distribution and the number of draws from the core RNG. This occurs for two reasons:

- The random values are simulated using a rejection-based method and so, on average, more than one value from the underlying RNG is required to generate an single draw.
- The number of bits required to generate a simulated value differs from the number of bits generated by the underlying RNG. For example, two 16-bit integer values can be simulated from a single draw of a 32-bit RNG.

Advancing the RNG state resets any pre-computed random numbers. This is required to ensure exact reproducibility.

method

`Philox.`**`jumped`**`(`*jumps=1*`)`

Returns a new bit generator with the state jumped

The state of the returned big generator is jumped as-if 2**(128 * jumps) random numbers have been generated.

**Parameters**

**jumps**

[integer, positive] Number of times to jump the state of the bit generator returned

**Returns**

**bit_generator**

[Philox] New instance of generator jumped iter times

## Extending

| | |
|---|---|
| *`cffi`* | CFFI interface |
| *`ctypes`* | ctypes interface |

attribute

`Philox.`**`cffi`**

CFFI interface

**Returns**

**interface**

[namedtuple] Named tuple containing CFFI wrapper

- state_address - Memory address of the state struct

- state - pointer to the state struct
- next_uint64 - function pointer to produce 64 bit integers
- next_uint32 - function pointer to produce 32 bit integers
- next_double - function pointer to produce doubles
- bitgen - pointer to the bit generator struct

attribute

`Philox.`**`ctypes`**

ctypes interface

**Returns**

**interface**

[namedtuple] Named tuple containing ctypes wrapper

- state_address - Memory address of the state struct
- state - pointer to the state struct
- next_uint64 - function pointer to produce 64 bit integers
- next_uint32 - function pointer to produce 32 bit integers
- next_double - function pointer to produce doubles
- bitgen - pointer to the bit generator struct

## SFC64 Small Fast Chaotic PRNG

**`class`** `numpy.random.`**`SFC64`**`(`*seed=None*`)`

BitGenerator for Chris Doty-Humphrey's Small Fast Chaotic PRNG.

**Parameters**

**seed**

[{None, int, array_like[ints], SeedSequence}, optional] A seed to initialize the *`BitGenerator`*. If None, then fresh, unpredictable entropy will be pulled from the OS. If an `int` or `array_like[ints]` is passed, then it will be passed to *`SeedSequence`* to derive the initial *`BitGenerator`* state. One may also pass in a *`SeedSequence`* instance.

### Notes

`SFC64` is a 256-bit implementation of Chris Doty-Humphrey's Small Fast Chaotic PRNG ([1]). `SFC64` has a few different cycles that one might be on, depending on the seed; the expected period will be about $2^{255}$ ([2]). `SFC64` incorporates a 64-bit counter which means that the absolute minimum cycle length is $2^{64}$ and that distinct seeds will not run into each other for at least $2^{64}$ iterations.

`SFC64` provides a capsule containing function pointers that produce doubles, and unsigned 32 and 64- bit integers. These are not directly consumable in Python and must be consumed by a `Generator` or similar object that supports low-level access.

**State and Seeding**

The `SFC64` state vector consists of 4 unsigned 64-bit values. The last is a 64-bit counter that increments by 1 each iteration.

The input seed is processed by *`SeedSequence`* to generate the first 3 values, then the `SFC64` algorithm is iterated a small number of times to mix.

**Compatibility Guarantee**

`SFC64` makes a guarantee that a fixed seed will always produce the same random integer stream.

### References

[1], [2]

## State

| | |
|---|---|
| *`state`* | Get or set the PRNG state |

attribute

`SFC64.`**`state`**

Get or set the PRNG state

**Returns**

**state**

[dict] Dictionary containing the information required to describe the state of the PRNG

## Extending

| | |
|---|---|
| *`cffi`* | CFFI interface |
| *`ctypes`* | ctypes interface |

attribute

`SFC64.`**`cffi`**

CFFI interface

**Returns**

**interface**

[namedtuple] Named tuple containing CFFI wrapper

- state_address - Memory address of the state struct
- state - pointer to the state struct
- next_uint64 - function pointer to produce 64 bit integers
- next_uint32 - function pointer to produce 32 bit integers
- next_double - function pointer to produce doubles
- bitgen - pointer to the bit generator struct

attribute

`SFC64.`**`ctypes`**

ctypes interface

**Returns**

**interface**

[namedtuple] Named tuple containing ctypes wrapper

- state_address - Memory address of the state struct
- state - pointer to the state struct
- next_uint64 - function pointer to produce 64 bit integers
- next_uint32 - function pointer to produce 32 bit integers
- next_double - function pointer to produce doubles
- bitgen - pointer to the bit generator struct

## Seeding and Entropy

A BitGenerator provides a stream of random values. In order to generate reproducible streams, BitGenerators support setting their initial state via a seed. All of the provided BitGenerators will take an arbitrary-sized non-negative integer, or a list of such integers, as a seed. BitGenerators need to take those inputs and process them into a high-quality internal state for the BitGenerator. All of the BitGenerators in numpy delegate that task to *`SeedSequence`*, which uses hashing techniques to ensure that even low-quality seeds generate high-quality initial states.

```
from numpy.random import PCG64

bg = PCG64(12345678903141592653589793)
```

*`SeedSequence`* is designed to be convenient for implementing best practices. We recommend that a stochastic program defaults to using entropy from the OS so that each run is different. The program should print out or log that entropy. In order to reproduce a past value, the program should allow the user to provide that value through some mechanism, a command-line argument is common, so that the user can then re-enter that entropy to reproduce the result. *`SeedSequence`* can take care of everything except for communicating with the user, which is up to you.

```
from numpy.random import PCG64, SeedSequence

# Get the user's seed somehow, maybe through `argparse`.
# If the user did not provide a seed, it should return `None`.
seed = get_user_seed()
ss = SeedSequence(seed)
print('seed = {}'.format(ss.entropy))
bg = PCG64(ss)
```

We default to using a 128-bit integer using entropy gathered from the OS. This is a good amount of entropy to initialize all of the generators that we have in numpy. We do not recommend using small seeds below 32 bits for general use. Using just a small set of seeds to instantiate larger state spaces means that there are some initial states that are impossible to reach. This creates some biases if everyone uses such values.

There will not be anything *wrong* with the results, per se; even a seed of 0 is perfectly fine thanks to the processing that *`SeedSequence`* does. If you just need *some* fixed value for unit tests or debugging, feel free to use whatever seed you like. But if you want to make inferences from the results or publish them, drawing from a larger set of seeds is good practice.

If you need to generate a good seed “offline”, then `SeedSequence().entropy` or using `secrets.randbits(128)` from the standard library are both convenient ways.

| | |
|---|---|
| *SeedSequence*([entropy, spawn_key, pool_size]) | SeedSequence mixes sources of entropy in a reproducible way to set the initial state for independent and very probably non-overlapping BitGenerators. |

**class** `numpy.random.`**`SeedSequence`**(*entropy=None*, ***, *spawn_key=()*, *pool_size=4*)

SeedSequence mixes sources of entropy in a reproducible way to set the initial state for independent and very probably non-overlapping BitGenerators.

Once the SeedSequence is instantiated, you can call the `generate_state` method to get an appropriately sized seed. Calling `spawn(n)` will create `n` SeedSequences that can be used to seed independent BitGenerators, i.e. for different threads.

**Parameters**

**entropy**

[{None, int, sequence[int]}, optional] The entropy for creating a `SeedSequence`.

**spawn_key**

[{(), sequence[int]}, optional] A third source of entropy, used internally when calling `SeedSequence.spawn`

**pool_size**

[{int}, optional] Size of the pooled entropy to store. Default is 4 to give a 128-bit entropy pool. 8 (for 256 bits) is another reasonable choice if working with larger PRNGs, but there is very little to be gained by selecting another value.

**n_children_spawned**

[{int}, optional] The number of children already spawned. Only pass this if reconstructing a `SeedSequence` from a serialized form.

## Notes

Best practice for achieving reproducible bit streams is to use the default `None` for the initial entropy, and then use `SeedSequence.entropy` to log/pickle the `entropy` for reproducibility:

```
>>> sq1 = np.random.SeedSequence()
>>> sq1.entropy
243799254704924441050048792905230269161  # random
>>> sq2 = np.random.SeedSequence(sq1.entropy)
>>> np.all(sq1.generate_state(10) == sq2.generate_state(10))
True
```

**Attributes**

**entropy**

**n_children_spawned**

**pool**

**pool_size**

**spawn_key**

**state**

## Methods

| | |
|---|---|
| *generate_state*(n_words[, dtype]) | Return the requested number of words for PRNG seeding. |
| *spawn*(n_children) | Spawn a number of child *SeedSequence* s by extending the `spawn_key`. |

method

`SeedSequence.`**`generate_state`**(*n_words*, *dtype=np.uint32*)

Return the requested number of words for PRNG seeding.

A BitGenerator should call this method in its constructor with an appropriate *n_words* parameter to properly seed itself.

**Parameters**

**n_words**

[int]

**dtype**

[np.uint32 or np.uint64, optional] The size of each word. This should only be either *uint32* or *uint64*. Strings (*'uint32'*, *'uint64'*) are fine. Note that requesting *uint64* will draw twice as many bits as *uint32* for the same *n_words*. This is a convenience for *BitGenerator's that express their states as 'uint64* arrays.

**Returns**

**state**

[uint32 or uint64 array, shape=(n_words,)]

method

`SeedSequence.`**`spawn`**(*n_children*)

Spawn a number of child *SeedSequence* s by extending the `spawn_key`.

**Parameters**

**n_children**

[int]

**Returns**

**seqs**

[list of *SeedSequence* s]

## 4.24.4 Features

### Parallel Random Number Generation

There are three strategies implemented that can be used to produce repeatable pseudo-random numbers across multiple processes (local or distributed).

#### `SeedSequence` spawning

*SeedSequence* implements an algorithm to process a user-provided seed, typically as an integer of some size, and to convert it into an initial state for a *BitGenerator*. It uses hashing techniques to ensure that low-quality seeds are turned into high quality initial states (at least, with very high probability).

For example, *MT19937* has a state consisting of 624 *uint32* integers. A naive way to take a 32-bit integer seed would be to just set the last element of the state to the 32-bit seed and leave the rest 0s. This is a valid state for *MT19937*, but not a good one. The Mersenne Twister algorithm suffers if there are too many 0s. Similarly, two adjacent 32-bit integer seeds (i.e. `12345` and `12346`) would produce very similar streams.

*SeedSequence* avoids these problems by using successions of integer hashes with good avalanche properties to ensure that flipping any bit in the input input has about a 50% chance of flipping any bit in the output. Two input seeds that are very close to each other will produce initial states that are very far from each other (with very high probability). It is also constructed in such a way that you can provide arbitrary-sized integers or lists of integers. *SeedSequence* will take all of the bits that you provide and mix them together to produce however many bits the consuming *BitGenerator* needs to initialize itself.

These properties together mean that we can safely mix together the usual user-provided seed with simple incrementing counters to get *BitGenerator* states that are (to very high probability) independent of each other. We can wrap this together into an API that is easy to use and difficult to misuse.

```
from numpy.random import SeedSequence, default_rng

ss = SeedSequence(12345)

# Spawn off 10 child SeedSequences to pass to child processes.
child_seeds = ss.spawn(10)
streams = [default_rng(s) for s in child_seeds]
```

Child *SeedSequence* objects can also spawn to make grandchildren, and so on. Each *SeedSequence* has its position in the tree of spawned *SeedSequence* objects mixed in with the user-provided seed to generate independent (with very high probability) streams.

```
grandchildren = child_seeds[0].spawn(4)
grand_streams = [default_rng(s) for s in grandchildren]
```

This feature lets you make local decisions about when and how to split up streams without coordination between processes. You do not have to preallocate space to avoid overlapping or request streams from a common global service. This general "tree-hashing" scheme is not unique to numpy but not yet widespread. Python has increasingly-flexible mechanisms for parallelization available, and this scheme fits in very well with that kind of use.

Using this scheme, an upper bound on the probability of a collision can be estimated if one knows the number of streams that you derive. *SeedSequence* hashes its inputs, both the seed and the spawn-tree-path, down to a 128-bit pool by default. The probability that there is a collision in that pool, pessimistically-estimated ([1]), will be about $n^2 * 2^{-128}$ where $n$ is the number of streams spawned. If a program uses an aggressive million streams, about $2^{20}$, then the probability that at least one pair of them are identical is about $2^{-88}$, which is in solidly-ignorable territory ([2]).

---

[1] The algorithm is carefully designed to eliminate a number of possible ways to collide. For example, if one only does one level of spawning, it is guaranteed that all states will be unique. But it's easier to estimate the naive upper bound on a napkin and take comfort knowing that the probability is actually lower.

[2] In this calculation, we can ignore the amount of numbers drawn from each stream. Each of the PRNGs we provide has some extra protection built

### Independent Streams

`Philox` is a counter-based RNG based which generates values by encrypting an incrementing counter using weak cryptographic primitives. The seed determines the key that is used for the encryption. Unique keys create unique, independent streams. `Philox` lets you bypass the seeding algorithm to directly set the 128-bit key. Similar, but different, keys will still create independent streams.

```
import secrets
from numpy.random import Philox

# 128-bit number as a seed
root_seed = secrets.getrandbits(128)
streams = [Philox(key=root_seed + stream_id) for stream_id in range(10)]
```

This scheme does require that you avoid reusing stream IDs. This may require coordination between the parallel processes.

### Jumping the BitGenerator state

`jumped` advances the state of the BitGenerator *as-if* a large number of random numbers have been drawn, and returns a new instance with this state. The specific number of draws varies by BitGenerator, and ranges from $2^{64}$ to $2^{128}$. Additionally, the *as-if* draws also depend on the size of the default random number produced by the specific BitGenerator. The BitGenerators that support `jumped`, along with the period of the BitGenerator, the size of the jump and the bits in the default unsigned random are listed below.

| BitGenerator | Period | Jump Size | Bits |
|---|---|---|---|
| MT19937 | $2^{19937}$ | $2^{128}$ | 32 |
| PCG64 | $2^{128}$ | $2^{127}$ ([3]) | 64 |
| Philox | $2^{256}$ | $2^{128}$ | 64 |

`jumped` can be used to produce long blocks which should be long enough to not overlap.

```
import secrets
from numpy.random import PCG64

seed = secrets.getrandbits(128)
blocked_rng = []
rng = PCG64(seed)
for i in range(10):
    blocked_rng.append(rng.jumped(i))
```

When using `jumped`, one does have to take care not to jump to a stream that was already used. In the above example, one could not later use `blocked_rng[0].jumped()` as it would overlap with `blocked_rng[1]`. Like with the independent streams, if the main process here wants to split off 10 more streams by jumping, then it needs to start with `range(10, 20)`, otherwise it would recreate the same streams. On the other hand, if you carefully construct the streams, then you are guaranteed to have streams that do not overlap.

---

in that avoids overlaps if the `SeedSequence` pools differ in the slightest bit. `PCG64` has $2^{127}$ separate cycles determined by the seed in addition to the position in the $2^{128}$ long period for each cycle, so one has to both get on or near the same cycle *and* seed a nearby position in the cycle. `Philox` has completely independent cycles determined by the seed. `SFC64` incorporates a 64-bit counter so every unique seed is at least $2^{64}$ iterations away from any other seed. And finally, `MT19937` has just an unimaginably huge period. Getting a collision internal to `SeedSequence` is the way a failure would be observed.

[3] The jump size is $(\phi - 1) * 2^{128}$ where $\phi$ is the golden ratio. As the jumps wrap around the period, the actual distances between neighboring streams will slowly grow smaller than the jump size, but using the golden ratio this way is a classic method of constructing a low-discrepancy sequence that spreads out the states around the period optimally. You will not be able to jump enough to make those distances small enough to overlap in your lifetime.

### Multithreaded Generation

The four core distributions (`random`, `standard_normal`, `standard_exponential`, and `standard_gamma`) all allow existing arrays to be filled using the `out` keyword argument. Existing arrays need to be contiguous and well-behaved (writable and aligned). Under normal circumstances, arrays created using the common constructors such as `numpy.empty` will satisfy these requirements.

This example makes use of Python 3 `concurrent.futures` to fill an array using multiple threads. Threads are long-lived so that repeated calls do not require any additional overheads from thread creation.

The random numbers generated are reproducible in the sense that the same seed will produce the same outputs, given that the number of threads does not change.

```
from numpy.random import default_rng, SeedSequence
import multiprocessing
import concurrent.futures
import numpy as np

class MultithreadedRNG:
    def __init__(self, n, seed=None, threads=None):
        if threads is None:
            threads = multiprocessing.cpu_count()
        self.threads = threads

        seq = SeedSequence(seed)
        self._random_generators = [default_rng(s)
                                   for s in seq.spawn(threads)]

        self.n = n
        self.executor = concurrent.futures.ThreadPoolExecutor(threads)
        self.values = np.empty(n)
        self.step = np.ceil(n / threads).astype(np.int_)

    def fill(self):
        def _fill(random_state, out, first, last):
            random_state.standard_normal(out=out[first:last])

        futures = {}
        for i in range(self.threads):
            args = (_fill,
                    self._random_generators[i],
                    self.values,
                    i * self.step,
                    (i + 1) * self.step)
            futures[self.executor.submit(*args)] = i
        concurrent.futures.wait(futures)

    def __del__(self):
        self.executor.shutdown(False)
```

The multithreaded random number generator can be used to fill an array. The `values` attributes shows the zero-value before the fill and the random value after.

```
In [2]: mrng = MultithreadedRNG(10000000, seed=12345)
   ...: print(mrng.values[-1])
Out[2]: 0.0

In [3]: mrng.fill()
```

(continues on next page)

(continued from previous page)

```
   ...: print(mrng.values[-1])
Out[3]: 2.4545724517479104
```

The time required to produce using multiple threads can be compared to the time required to generate using a single thread.

```
In [4]: print(mrng.threads)
   ...: %timeit mrng.fill()

Out[4]: 4
   ...: 32.8 ms ± 2.71 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

The single threaded call directly uses the BitGenerator.

```
In [5]: values = np.empty(10000000)
   ...: rg = default_rng()
   ...: %timeit rg.standard_normal(out=values)

Out[5]: 99.6 ms ± 222 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

The gains are substantial and the scaling is reasonable even for arrays that are only moderately large. The gains are even larger when compared to a call that does not use an existing array due to array creation overhead.

```
In [6]: rg = default_rng()
   ...: %timeit rg.standard_normal(10000000)

Out[6]: 125 ms ± 309 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

Note that if *threads* is not set by the user, it will be determined by *multiprocessing.cpu_count()*.

```
In [7]: # simulate the behavior for `threads=None`, if the machine had only one thread
   ...: mrng = MultithreadedRNG(10000000, seed=12345, threads=1)
   ...: print(mrng.values[-1])
Out[7]: 1.1800150052158556
```

### What's New or Different

**Warning:** The Box-Muller method used to produce NumPy's normals is no longer available in *`Generator`*. It is not possible to reproduce the exact random values using `Generator` for the normal distribution or any other distribution that relies on the normal such as the *`Generator.gamma`* or *`Generator.standard_t`*. If you require bitwise backward compatible streams, use *`RandomState`*, i.e., *`RandomState.gamma`* or *`RandomState.standard_t`*.

Quick comparison of legacy mtrand to the new *`Generator`*

| Feature | Older Equivalent | Notes |
|---|---|---|
| `Generator` | `RandomState` | `Generator` requires a stream source, called a `BitGenerator` A number of these are provided. `RandomState` uses the Mersenne Twister `MT19937` by default, but can also be instantiated with any BitGenerator. |
| `random` | `random_sample`,<br>`rand` | Access the values in a BitGenerator, convert them to `float64` in the interval `[0.0.`, “ 1.0)“. In addition to the `size` kwarg, now supports `dtype='d'` or `dtype='f'`, and an `out` kwarg to fill a user- supplied array.<br>Many other distributions are also supported. |
| `integers` | `randint`,<br>`random_integers` | Use the `endpoint` kwarg to adjust the inclusion or exclution of the `high` interval endpoint |

And in more detail:

- Simulate from the complex normal distribution (*complex_normal*)
- The normal, exponential and gamma generators use 256-step Ziggurat methods which are 2-10 times faster than NumPy's default implementation in `standard_normal`, `standard_exponential` or `standard_gamma`.

```
In [1]: from  numpy.random import Generator, PCG64

In [2]: import numpy.random

In [3]: rg = Generator(PCG64())

In [4]: %timeit -n 1 rg.standard_normal(100000)
   ...: %timeit -n 1 numpy.random.standard_normal(100000)
   ...:
1.08 ms +- 10.3 us per loop (mean +- std. dev. of 7 runs, 1 loop each)
2.08 ms +- 118 us per loop (mean +- std. dev. of 7 runs, 1 loop each)
```

```
In [5]: %timeit -n 1 rg.standard_exponential(100000)
   ...: %timeit -n 1 numpy.random.standard_exponential(100000)
   ...:
447 us +- 6.01 us per loop (mean +- std. dev. of 7 runs, 1 loop each)
1.45 ms +- 4.65 us per loop (mean +- std. dev. of 7 runs, 1 loop each)
```

```
In [6]: %timeit -n 1 rg.standard_gamma(3.0, 100000)
   ...: %timeit -n 1 numpy.random.standard_gamma(3.0, 100000)
   ...:
2.12 ms +- 12.7 us per loop (mean +- std. dev. of 7 runs, 1 loop each)
4.03 ms +- 62.9 us per loop (mean +- std. dev. of 7 runs, 1 loop each)
```

- `integers` is now the canonical way to generate integer random numbers from a discrete uniform distribution. The `rand` and `randn` methods are only available through the legacy `RandomState`. This replaces both `randint` and the deprecated `random_integers`.
- The Box-Muller method used to produce NumPy's normals is no longer available.
- All bit generators can produce doubles, uint64s and uint32s via CTypes (`ctypes`) and CFFI (`cffi`). This allows these bit generators to be used in numba.
- The bit generators can be used in downstream projects via Cython.
- Optional `dtype` argument that accepts `np.float32` or `np.float64` to produce either single or double prevision uniform random variables for select distributions

  - Uniforms (*`random`* and *`integers`*)
  - Normals (*`standard_normal`*)
  - Standard Gammas (*`standard_gamma`*)
  - Standard Exponentials (*`standard_exponential`*)

```
In [7]: rg = Generator(PCG64(0))

In [8]: rg.random(3, dtype='d')
Out[8]: array([0.63696169, 0.26978671, 0.04097352])

In [9]: rg.random(3, dtype='f')
Out[9]: array([0.07524014, 0.01652753, 0.17526722], dtype=float32)
```

- Optional `out` argument that allows existing arrays to be filled for select distributions
  - Uniforms (*`random`*)
  - Normals (*`standard_normal`*)
  - Standard Gammas (*`standard_gamma`*)
  - Standard Exponentials (*`standard_exponential`*)

  This allows multithreading to fill large arrays in chunks using suitable BitGenerators in parallel.

```
In [10]: existing = np.zeros(4)

In [11]: rg.random(out=existing[:2])
Out[11]: array([0.91275558, 0.60663578])

In [12]: print(existing)
[0.91275558 0.60663578 0.         0.        ]
```

- Optional `axis` argument for methods like *`choice`*, *`permutation`* and *`shuffle`* that controls which axis an operation is performed over for multi-dimensional arrays.

```
In [13]: rg = Generator(PCG64(123456789))

In [14]: a = np.arange(12).reshape((3, 4))

In [15]: a
Out[15]: 
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])

In [16]: rg.choice(a, axis=1, size=5)
Out[16]: 
array([[ 3,  0,  2,  3,  1],
       [ 7,  4,  6,  7,  5],
       [11,  8, 10, 11,  9]])

In [17]: rg.shuffle(a, axis=1)        # Shuffle in-place

In [18]: a
Out[18]: 
array([[ 3,  1,  2,  0],
```

(continues on next page)

(continued from previous page)

```
       [ 7,  5,  6,  4],
       [11,  9, 10,  8]])
```

## Performance

### Recommendation

The recommended generator for general use is `PCG64`. It is statistically high quality, full-featured, and fast on most platforms, but somewhat slow when compiled for 32-bit processes.

`Philox` is fairly slow, but its statistical properties have very high quality, and it is easy to get assuredly-independent stream by using unique keys. If that is the style you wish to use for parallel streams, or you are porting from another system that uses that style, then `Philox` is your choice.

`SFC64` is statistically high quality and very fast. However, it lacks jumpability. If you are not using that capability and want lots of speed, even on 32-bit processes, this is your choice.

`MT19937` fails some statistical tests and is not especially fast compared to modern PRNGs. For these reasons, we mostly do not recommend using it on its own, only through the legacy `RandomState` for reproducing old results. That said, it has a very long history as a default in many systems.

### Timings

The timings below are the time in ns to produce 1 random value from a specific distribution. The original `MT19937` generator is much slower since it requires 2 32-bit values to equal the output of the faster generators.

Integer performance has a similar ordering.

The pattern is similar for other, more complex generators. The normal performance of the legacy `RandomState` generator is much lower than the other since it uses the Box-Muller transformation rather than the Ziggurat generator. The performance gap for Exponentials is also large due to the cost of computing the log function to invert the CDF. The column labeled MT19973 is used the same 32-bit generator as `RandomState` but produces random values using `Generator`.

| | MT19937 | PCG64 | Philox | SFC64 | RandomState |
|---|---|---|---|---|---|
| 32-bit Unsigned Ints | 3.2 | 2.7 | 4.9 | 2.7 | 3.2 |
| 64-bit Unsigned Ints | 5.6 | 3.7 | 6.3 | 2.9 | 5.7 |
| Uniforms | 7.3 | 4.1 | 8.1 | 3.1 | 7.3 |
| Normals | 13.1 | 10.2 | 13.5 | 7.8 | 34.6 |
| Exponentials | 7.9 | 5.4 | 8.5 | 4.1 | 40.3 |
| Gammas | 34.8 | 28.0 | 34.7 | 25.1 | 58.1 |
| Binomials | 25.0 | 21.4 | 26.1 | 19.5 | 25.2 |
| Laplaces | 45.1 | 40.7 | 45.5 | 38.1 | 45.6 |
| Poissons | 67.6 | 52.4 | 69.2 | 46.4 | 78.1 |

The next table presents the performance in percentage relative to values generated by the legacy generator, `RandomState(MT19937())`. The overall performance was computed using a geometric mean.

| | MT19937 | PCG64 | Philox | SFC64 |
|---|---|---|---|---|
| 32-bit Unsigned Ints | 101 | 121 | 67 | 121 |
| 64-bit Unsigned Ints | 102 | 156 | 91 | 199 |
| Uniforms | 100 | 179 | 90 | 235 |
| Normals | 263 | 338 | 257 | 443 |
| Exponentials | 507 | 752 | 474 | 985 |
| Gammas | 167 | 207 | 167 | 231 |
| Binomials | 101 | 118 | 96 | 129 |
| Laplaces | 101 | 112 | 100 | 120 |
| Poissons | 116 | 149 | 113 | 168 |
| Overall | 144 | 192 | 132 | 225 |

**Note:** All timings were taken using Linux on an i5-3570 processor.

### Performance on different Operating Systems

Performance differs across platforms due to compiler and hardware availability (e.g., register width) differences. The default bit generator has been chosen to perform well on 64-bit platforms. Performance on 32-bit operating systems is very different.

The values reported are normalized relative to the speed of MT19937 in each table. A value of 100 indicates that the performance matches the MT19937. Higher values indicate improved performance. These values cannot be compared across tables.

#### 64-bit Linux

| Distribution | MT19937 | PCG64 | Philox | SFC64 |
|---|---|---|---|---|
| 32-bit Unsigned Int | 100 | 119.8 | 67.7 | 120.2 |
| 64-bit Unsigned Int | 100 | 152.9 | 90.8 | 213.3 |
| Uniforms | 100 | 179.0 | 87.0 | 232.0 |
| Normals | 100 | 128.5 | 99.2 | 167.8 |
| Exponentials | 100 | 148.3 | 93.0 | 189.3 |
| **Overall** | 100 | 144.3 | 86.8 | 180.0 |

#### 64-bit Windows

The relative performance on 64-bit Linux and 64-bit Windows is broadly similar.

| Distribution | MT19937 | PCG64 | Philox | SFC64 |
|---|---|---|---|---|
| 32-bit Unsigned Int | 100 | 129.1 | 35.0 | 135.0 |
| 64-bit Unsigned Int | 100 | 146.9 | 35.7 | 176.5 |
| Uniforms | 100 | 165.0 | 37.0 | 192.0 |
| Normals | 100 | 128.5 | 48.5 | 158.0 |
| Exponentials | 100 | 151.6 | 39.0 | 172.8 |
| **Overall** | 100 | 143.6 | 38.7 | 165.7 |

### 32-bit Windows

The performance of 64-bit generators on 32-bit Windows is much lower than on 64-bit operating systems due to register width. MT19937, the generator that has been in NumPy since 2005, operates on 32-bit integers.

| Distribution | MT19937 | PCG64 | Philox | SFC64 |
|---|---|---|---|---|
| 32-bit Unsigned Int | 100 | 30.5 | 21.1 | 77.9 |
| 64-bit Unsigned Int | 100 | 26.3 | 19.2 | 97.0 |
| Uniforms | 100 | 28.0 | 23.0 | 106.0 |
| Normals | 100 | 40.1 | 31.3 | 112.6 |
| Exponentials | 100 | 33.7 | 26.3 | 109.8 |
| **Overall** | 100 | 31.4 | 23.8 | 99.8 |

**Note:** Linux timings used Ubuntu 18.04 and GCC 7.4. Windows timings were made on Windows 10 using Microsoft C/C++ Optimizing Compiler Version 19 (Visual Studio 2015). All timings were produced on an i5-3570 processor.

### Cython API for random

Typed versions of many of the *`Generator`* and *`BitGenerator`* methods as well as the classes themselves can be accessed directly from Cython via

```
cimport numpy.random
```

### C API for random

Access to various distributions is available via Cython or C-wrapper libraries like CFFI. All the functions accept a *`bitgen_t`* as their first argument.

**`bitgen_t`**

The *`bitgen_t`* holds the current state of the BitGenerator and pointers to functions that return standard C types while advancing the state.

```
struct bitgen:
    void *state
    npy_uint64 (*next_uint64)(void *st) nogil
    uint32_t (*next_uint32)(void *st) nogil
    double (*next_double)(void *st) nogil
    npy_uint64 (*next_raw)(void *st) nogil

ctypedef bitgen bitgen_t
```

See *Extending* for examples of using these functions.

The functions are named with the following conventions:

- "standard" refers to the reference values for any parameters. For instance "standard_uniform" means a uniform distribution on the interval `0.0` to `1.0`
- "fill" functions will fill the provided `out` with `cnt` values.
- The functions without "standard" in their name require additional parameters to describe the distributions.

- `zig` in the name are based on a ziggurat lookup algorithm is used instead of calculating the `log`, which is significantly faster. The non-ziggurat variants are used in corner cases and for legacy compatibility.

double **`random_standard_uniform`** (*bitgen_t *bitgen_state*)

void **`random_standard_uniform_fill`** (*bitgen_t* bitgen_state*, *npy_intp cnt*, double **out*)

double **`random_standard_exponential`** (*bitgen_t *bitgen_state*)

void **`random_standard_exponential_fill`** (*bitgen_t *bitgen_state*, *npy_intp cnt*, double **out*)

double **`random_standard_normal`** (*bitgen_t* bitgen_state*)

void **`random_standard_normal_fill`** (*bitgen_t *bitgen_state*, *npy_intp count*, double **out*)

void **`random_standard_normal_fill_f`** (*bitgen_t *bitgen_state*, *npy_intp count*, float **out*)

double **`random_standard_gamma`** (*bitgen_t *bitgen_state*, double *shape*)

float **`random_standard_uniform_f`** (*bitgen_t *bitgen_state*)

void **`random_standard_uniform_fill_f`** (*bitgen_t* bitgen_state*, *npy_intp cnt*, float **out*)

float **`random_standard_exponential_f`** (*bitgen_t *bitgen_state*)

void **`random_standard_exponential_fill_f`** (*bitgen_t *bitgen_state*, *npy_intp cnt*, float **out*)

float **`random_standard_normal_f`** (*bitgen_t* bitgen_state*)

float **`random_standard_gamma_f`** (*bitgen_t *bitgen_state*, float *shape*)

double **`random_normal`** (*bitgen_t *bitgen_state*, double *loc*, double *scale*)

double **`random_gamma`** (*bitgen_t *bitgen_state*, double *shape*, double *scale*)

float **`random_gamma_f`** (*bitgen_t *bitgen_state*, float *shape*, float *scale*)

double **`random_exponential`** (*bitgen_t *bitgen_state*, double *scale*)

double **`random_uniform`** (*bitgen_t *bitgen_state*, double *lower*, double *range*)

double **`random_beta`** (*bitgen_t *bitgen_state*, double *a*, double *b*)

double **`random_chisquare`** (*bitgen_t *bitgen_state*, double *df*)

double **`random_f`** (*bitgen_t *bitgen_state*, double *dfnum*, double *dfden*)

double **`random_standard_cauchy`** (*bitgen_t *bitgen_state*)

double **`random_pareto`** (*bitgen_t *bitgen_state*, double *a*)

double **`random_weibull`** (*bitgen_t *bitgen_state*, double *a*)

double **`random_power`** (*bitgen_t *bitgen_state*, double *a*)

double **`random_laplace`** (*bitgen_t *bitgen_state*, double *loc*, double *scale*)

double **`random_gumbel`** (*bitgen_t *bitgen_state*, double *loc*, double *scale*)

double **`random_logistic`** (*bitgen_t *bitgen_state*, double *loc*, double *scale*)

double **`random_lognormal`** (*bitgen_t *bitgen_state*, double *mean*, double *sigma*)

double **`random_rayleigh`** (*bitgen_t *bitgen_state*, double *mode*)

double **`random_standard_t`** (*bitgen_t *bitgen_state*, double *df*)

double **`random_noncentral_chisquare`** (*bitgen_t *bitgen_state*, double *df*, double *nonc*)

double **`random_noncentral_f`** (*bitgen_t *bitgen_state*, double *dfnum*, double *dfden*, double *nonc*) double **random_wald** (*bitgen_t *bitgen_state*, double *mean*, double *scale*)

double **random_vonmises** (*bitgen_t *bitgen_state*, double *mu*, double *kappa*)

double **random_triangular** (*bitgen_t *bitgen_state*, double *left*, double *mode*, double *right*)

*npy_int64* **random_poisson** (*bitgen_t *bitgen_state*, double *lam*)

*npy_int64* **random_negative_binomial** (*bitgen_t *bitgen_state*, double *n*, double *p*)

**binomial_t**

```
typedef struct s_binomial_t {
  int has_binomial; /* !=0: following parameters initialized for binomial */
  double psave;
  RAND_INT_TYPE nsave;
  double r;
  double q;
  double fm;
  RAND_INT_TYPE m;
  double p1;
  double xm;
  double xl;
  double xr;
  double c;
  double laml;
  double lamr;
  double p2;
  double p3;
  double p4;
} binomial_t;
```

*npy_int64* **random_binomial** (*bitgen_t *bitgen_state*, double *p*, *npy_int64 n*, *binomial_t *binomial*)

*npy_int64* **random_logseries** (*bitgen_t *bitgen_state*, double *p*)

*npy_int64* **random_geometric_search** (*bitgen_t *bitgen_state*, double *p*)

*npy_int64* **random_geometric_inversion** (*bitgen_t *bitgen_state*, double *p*)

*npy_int64* **random_geometric** (*bitgen_t *bitgen_state*, double *p*)

*npy_int64* **random_zipf** (*bitgen_t *bitgen_state*, double *a*)

*npy_int64* **random_hypergeometric** (*bitgen_t *bitgen_state*, *npy_int64 good*, *npy_int64 bad*, *npy_int64 sample*)

*npy_uint64* **random_interval** (*bitgen_t *bitgen_state*, *npy_uint64 max*)

void **random_multinomial** (*bitgen_t *bitgen_state*, *npy_int64 n*, *npy_int64 *mnix*, double **pix*, *npy_intp d*, *binomial_t *binomial*)

int **random_multivariate_hypergeometric_count** (*bitgen_t *bitgen_state*, *npy_int64 total*, size_t *num_colors*, *npy_int64 *colors*, *npy_int64 nsample*, size_t *num_variates*, *npy_int64 *variates*)

void **random_multivariate_hypergeometric_marginals** (*bitgen_t *bitgen_state*, *npy_int64 total*, size_t *num_colors*, *npy_int64 *colors*, *npy_int64 nsample*, size_t *num_variates*, *npy_int64 *variates*)

Generate a single integer

*npy_int64* **random_positive_int64** (*bitgen_t *bitgen_state*)

*npy_int32* **random_positive_int32** (*bitgen_t *bitgen_state*)

*npy_int64* **random_positive_int** (*bitgen_t *bitgen_state*)

*npy_uint64* **random_uint** (*bitgen_t *bitgen_state*)

Generate random uint64 numbers in closed interval [off, off + rng].

*npy_uint64* **random_bounded_uint64** (*bitgen_t *bitgen_state*, *npy_uint64 off*, *npy_uint64 rng*, *npy_uint64 mask*, bint *use_masked*)

## Extending

The BitGenerators have been designed to be extendable using standard tools for high-performance Python – numba and Cython. The `Generator` object can also be used with user-provided BitGenerators as long as these export a small set of required functions.

### Numba

Numba can be used with either CTypes or CFFI. The current iteration of the BitGenerators all export a small set of functions through both interfaces.

This example shows how numba can be used to produce gaussian samples using a pure Python implementation which is then compiled. The random numbers are provided by `ctypes.next_double`.

```
import numpy as np
import numba as nb

from numpy.random import PCG64
from timeit import timeit

bit_gen = PCG64()
next_d = bit_gen.cffi.next_double
state_addr = bit_gen.cffi.state_address

def normals(n, state):
    out = np.empty(n)
    for i in range((n + 1) // 2):
        x1 = 2.0 * next_d(state) - 1.0
        x2 = 2.0 * next_d(state) - 1.0
        r2 = x1 * x1 + x2 * x2
        while r2 >= 1.0 or r2 == 0.0:
            x1 = 2.0 * next_d(state) - 1.0
            x2 = 2.0 * next_d(state) - 1.0
            r2 = x1 * x1 + x2 * x2
        f = np.sqrt(-2.0 * np.log(r2) / r2)
        out[2 * i] = f * x1
        if 2 * i + 1 < n:
            out[2 * i + 1] = f * x2
    return out

# Compile using Numba
normalsj = nb.jit(normals, nopython=True)
# Must use state address not state with numba
n = 10000

def numbacall():
```

(continues on next page)

(continued from previous page)

```
    return normalsj(n, state_addr)

rg = np.random.Generator(PCG64())

def numpycall():
    return rg.normal(size=n)

# Check that the functions work
r1 = numbacall()
r2 = numpycall()
assert r1.shape == (n,)
assert r1.shape == r2.shape

t1 = timeit(numbacall, number=1000)
print('{:.2f} secs for {} PCG64 (Numba/PCG64) gaussian randoms'.format(t1, n))
t2 = timeit(numpycall, number=1000)
print('{:.2f} secs for {} PCG64 (NumPy/PCG64) gaussian randoms'.format(t2, n))
```

Both CTypes and CFFI allow the more complicated distributions to be used directly in Numba after compiling the file distributions.c into a `DLL` or `so`. An example showing the use of a more complicated distribution is in the *examples* section below.

### Cython

Cython can be used to unpack the `PyCapsule` provided by a BitGenerator. This example uses *`PCG64`* and the example from above. The usual caveats for writing high-performance code using Cython – removing bounds checks and wrap around, providing array alignment information – still apply.

```
#!/usr/bin/env python3
#cython: language_level=3
"""
This file shows how the to use a BitGenerator to create a distribution.
"""
import numpy as np
cimport numpy as np
cimport cython
from cpython.pycapsule cimport PyCapsule_IsValid, PyCapsule_GetPointer
from libc.stdint cimport uint16_t, uint64_t
from numpy.random cimport bitgen_t
from numpy.random import PCG64
from numpy.random.c_distributions cimport (
      random_standard_uniform_fill, random_standard_uniform_fill_f)


@cython.boundscheck(False)
@cython.wraparound(False)
def uniforms(Py_ssize_t n):
    """
    Create an array of `n` uniformly distributed doubles.
    A 'real' distribution would want to process the values into
    some non-uniform distribution
    """
    cdef Py_ssize_t i
    cdef bitgen_t *rng
    cdef const char *capsule_name = "BitGenerator"
    cdef double[::1] random_values
```

(continues on next page)

(continued from previous page)

```
    x = PCG64()
    capsule = x.capsule
    # Optional check that the capsule if from a BitGenerator
    if not PyCapsule_IsValid(capsule, capsule_name):
        raise ValueError("Invalid pointer to anon_func_state")
    # Cast the pointer
    rng = <bitgen_t *> PyCapsule_GetPointer(capsule, capsule_name)
    random_values = np.empty(n, dtype='float64')
    with x.lock, nogil:
        for i in range(n):
            # Call the function
            random_values[i] = rng.next_double(rng.state)
    randoms = np.asarray(random_values)

    return randoms
```

The BitGenerator can also be directly accessed using the members of the `bitgen_t` struct.

```
@cython.boundscheck(False)
@cython.wraparound(False)
def uint10_uniforms(Py_ssize_t n):
    """Uniform 10 bit integers stored as 16-bit unsigned integers"""
    cdef Py_ssize_t i
    cdef bitgen_t *rng
    cdef const char *capsule_name = "BitGenerator"
    cdef uint16_t[::1] random_values
    cdef int bits_remaining
    cdef int width = 10
    cdef uint64_t buff, mask = 0x3FF

    x = PCG64()
    capsule = x.capsule
    if not PyCapsule_IsValid(capsule, capsule_name):
        raise ValueError("Invalid pointer to anon_func_state")
    rng = <bitgen_t *> PyCapsule_GetPointer(capsule, capsule_name)
    random_values = np.empty(n, dtype='uint16')
    # Best practice is to release GIL and acquire the lock
    bits_remaining = 0
    with x.lock, nogil:
        for i in range(n):
            if bits_remaining < width:
                buff = rng.next_uint64(rng.state)
            random_values[i] = buff & mask
            buff >>= width

    randoms = np.asarray(random_values)
    return randoms
```

Cython can be used to directly access the functions in `numpy/random/c_distributions.pxd`. This requires linking with the `npyrandom` library located in `numpy/random/lib`.

```
def uniforms_ex(bit_generator, Py_ssize_t n, dtype=np.float64):
    """
    Create an array of `n` uniformly distributed doubles via a "fill" function.

```

(continues on next page)

(continued from previous page)

```
    A 'real' distribution would want to process the values into
    some non-uniform distribution

    Parameters
    ----------
    bit_generator: BitGenerator instance
    n: int
        Output vector length
    dtype: {str, dtype}, optional
        Desired dtype, either 'd' (or 'float64') or 'f' (or 'float32'). The
        default dtype value is 'd'
    """
    cdef Py_ssize_t i
    cdef bitgen_t *rng
    cdef const char *capsule_name = "BitGenerator"
    cdef np.ndarray randoms

    capsule = bit_generator.capsule
    # Optional check that the capsule if from a BitGenerator
    if not PyCapsule_IsValid(capsule, capsule_name):
        raise ValueError("Invalid pointer to anon_func_state")
    # Cast the pointer
    rng = <bitgen_t *> PyCapsule_GetPointer(capsule, capsule_name)

    _dtype = np.dtype(dtype)
    randoms = np.empty(n, dtype=_dtype)
    if _dtype == np.float32:
        with bit_generator.lock:
            random_standard_uniform_fill_f(rng, n, <float*>np.PyArray_DATA(randoms))
    elif _dtype == np.float64:
        with bit_generator.lock:
            random_standard_uniform_fill(rng, n, <double*>np.PyArray_DATA(randoms))
    else:
        raise TypeError('Unsupported dtype %r for random' % _dtype)
    return randoms
```

See *Extending numpy.random via Cython* for the complete listings of these examples and a minimal `setup.py` to build the c-extension modules.

### CFFI

CFFI can be used to directly access the functions in `include/numpy/random/distributions.h`. Some “massaging” of the header file is required:

```
"""
Use cffi to access any of the underlying C functions from distributions.h
"""
import os
import numpy as np
import cffi
from .parse import parse_distributions_h
ffi = cffi.FFI()

inc_dir = os.path.join(np.get_include(), 'numpy')

# Basic numpy types
ffi.cdef('''
```

(continues on next page)

(continued from previous page)

```
    typedef intptr_t npy_intp;
    typedef unsigned char npy_bool;

''')

parse_distributions_h(ffi, inc_dir)
```

Once the header is parsed by `ffi.cdef`, the functions can be accessed directly from the `_generator` shared object, using the `BitGenerator.cffi` interface.

```

# Compare the distributions.h random_standard_normal_fill to
# Generator.standard_random
bit_gen = np.random.PCG64()
rng = np.random.Generator(bit_gen)
state = bit_gen.state

interface = rng.bit_generator.cffi
n = 100
vals_cffi = ffi.new('double[%d]' % n)
lib.random_standard_normal_fill(interface.bit_generator, n, vals_cffi)

# reset the state
bit_gen.state = state

vals = rng.standard_normal(n)

for i in range(n):
    assert vals[i] == vals_cffi[i]
```

### New Bit Generators

*`Generator`* can be used with user-provided *`BitGenerator`*s. The simplest way to write a new BitGenerator is to examine the pyx file of one of the existing BitGenerators. The key structure that must be provided is the `capsule` which contains a `PyCapsule` to a struct pointer of type `bitgen_t`,

```
typedef struct bitgen {
  void *state;
  uint64_t (*next_uint64)(void *st);
  uint32_t (*next_uint32)(void *st);
  double (*next_double)(void *st);
  uint64_t (*next_raw)(void *st);
} bitgen_t;
```

which provides 5 pointers. The first is an opaque pointer to the data structure used by the BitGenerators. The next three are function pointers which return the next 64- and 32-bit unsigned integers, the next random double and the next raw value. This final function is used for testing and so can be set to the next 64-bit unsigned integer function if not needed. Functions inside `Generator` use this structure as in

```
bitgen_state->next_uint64(bitgen_state->state)
```

### Examples

### Extending via Numba

```
import numpy as np
import numba as nb

from numpy.random import PCG64
from timeit import timeit

bit_gen = PCG64()
next_d = bit_gen.cffi.next_double
state_addr = bit_gen.cffi.state_address

def normals(n, state):
    out = np.empty(n)
    for i in range((n + 1) // 2):
        x1 = 2.0 * next_d(state) - 1.0
        x2 = 2.0 * next_d(state) - 1.0
        r2 = x1 * x1 + x2 * x2
        while r2 >= 1.0 or r2 == 0.0:
            x1 = 2.0 * next_d(state) - 1.0
            x2 = 2.0 * next_d(state) - 1.0
            r2 = x1 * x1 + x2 * x2
        f = np.sqrt(-2.0 * np.log(r2) / r2)
        out[2 * i] = f * x1
        if 2 * i + 1 < n:
            out[2 * i + 1] = f * x2
    return out

# Compile using Numba
normalsj = nb.jit(normals, nopython=True)
# Must use state address not state with numba
n = 10000

def numbacall():
    return normalsj(n, state_addr)

rg = np.random.Generator(PCG64())

def numpycall():
    return rg.normal(size=n)

# Check that the functions work
r1 = numbacall()
r2 = numpycall()
assert r1.shape == (n,)
assert r1.shape == r2.shape

t1 = timeit(numbacall, number=1000)
print('{:.2f} secs for {} PCG64 (Numba/PCG64) gaussian randoms'.format(t1, n))
t2 = timeit(numpycall, number=1000)
print('{:.2f} secs for {} PCG64 (NumPy/PCG64) gaussian randoms'.format(t2, n))

# example 2

next_u32 = bit_gen.ctypes.next_uint32
```

(continues on next page)

(continued from previous page)

```
ctypes_state = bit_gen.ctypes.state

@nb.jit(nopython=True)
def bounded_uint(lb, ub, state):
    mask = delta = ub - lb
    mask |= mask >> 1
    mask |= mask >> 2
    mask |= mask >> 4
    mask |= mask >> 8
    mask |= mask >> 16

    val = next_u32(state) & mask
    while val > delta:
        val = next_u32(state) & mask

    return lb + val


print(bounded_uint(323, 2394691, ctypes_state.value))


@nb.jit(nopython=True)
def bounded_uints(lb, ub, n, state):
    out = np.empty(n, dtype=np.uint32)
    for i in range(n):
        out[i] = bounded_uint(lb, ub, state)


bounded_uints(323, 2394691, 10000000, ctypes_state.value)
```

### Extending via Numba and CFFI

```
r"""
Building the required library in this example requires a source distribution
of NumPy or clone of the NumPy git repository since distributions.c is not
included in binary distributions.

On *nix, execute in numpy/random/src/distributions

export ${PYTHON_VERSION}=3.8 # Python version
export PYTHON_INCLUDE=#path to Python's include folder, usually \
    ${PYTHON_HOME}/include/python${PYTHON_VERSION}m
export NUMPY_INCLUDE=#path to numpy's include folder, usually \
    ${PYTHON_HOME}/lib/python${PYTHON_VERSION}/site-packages/numpy/core/include
gcc -shared -o libdistributions.so -fPIC distributions.c \
    -I${NUMPY_INCLUDE} -I${PYTHON_INCLUDE}
mv libdistributions.so ../../_examples/numba/

On Windows

rem PYTHON_HOME and PYTHON_VERSION are setup dependent, this is an example
set PYTHON_HOME=c:\Anaconda
```

(continues on next page)

(continued from previous page)

```
set PYTHON_VERSION=38
cl.exe /LD .\distributions.c -DDLL_EXPORT \
    -I%PYTHON_HOME%\lib\site-packages\numpy\core\include \
    -I%PYTHON_HOME%\include %PYTHON_HOME%\libs\python%PYTHON_VERSION%.lib
move distributions.dll ../../_examples/numba/
"""
import os

import numba as nb
import numpy as np
from cffi import FFI

from numpy.random import PCG64

ffi = FFI()
if os.path.exists('./distributions.dll'):
    lib = ffi.dlopen('./distributions.dll')
elif os.path.exists('./libdistributions.so'):
    lib = ffi.dlopen('./libdistributions.so')
else:
    raise RuntimeError('Required DLL/so file was not found.')

ffi.cdef("""
double random_standard_normal(void *bitgen_state);
""")
x = PCG64()
xffi = x.cffi
bit_generator = xffi.bit_generator

random_standard_normal = lib.random_standard_normal


def normals(n, bit_generator):
    out = np.empty(n)
    for i in range(n):
        out[i] = random_standard_normal(bit_generator)
    return out


normalsj = nb.jit(normals, nopython=True)

# Numba requires a memory address for void *
# Can also get address from x.ctypes.bit_generator.value
bit_generator_address = int(ffi.cast('uintptr_t', bit_generator))

norm = normalsj(1000, bit_generator_address)
print(norm[:12])
```

## Extending `numpy.random` via Cython

### setup.py

```
#!/usr/bin/env python3
"""
Build the Cython demonstrations of low-level access to NumPy random

Usage: python setup.py build_ext -i
"""

import numpy as np
from distutils.core import setup
from Cython.Build import cythonize
from setuptools.extension import Extension
from os.path import join, dirname

path = dirname(__file__)
src_dir = join(dirname(path), '..', 'src')
defs = [('NPY_NO_DEPRECATED_API', 0)]
inc_path = np.get_include()
# not so nice. We need the random/lib library from numpy
lib_path = join(np.get_include(), '..', '..', 'random', 'lib')

extending = Extension("extending",
                      sources=[join(path, 'extending.pyx')],
                      include_dirs=[
                            np.get_include(),
                            join(path, '..', '..')
                        ],
                      define_macros=defs,
                      )
distributions = Extension("extending_distributions",
                          sources=[join(path, 'extending_distributions.pyx')],
                          include_dirs=[inc_path],
                          library_dirs=[lib_path],
                          libraries=['npyrandom'],
                          define_macros=defs,
                         )

extensions = [extending, distributions]

setup(
    ext_modules=cythonize(extensions)
)
```

### extending.pyx

```
#!/usr/bin/env python3
#cython: language_level=3

from libc.stdint cimport uint32_t
from cpython.pycapsule cimport PyCapsule_IsValid, PyCapsule_GetPointer

import numpy as np
cimport numpy as np
cimport cython

from numpy.random cimport bitgen_t
from numpy.random import PCG64

np.import_array()


@cython.boundscheck(False)
@cython.wraparound(False)
def uniform_mean(Py_ssize_t n):
    cdef Py_ssize_t i
    cdef bitgen_t *rng
    cdef const char *capsule_name = "BitGenerator"
    cdef double[::1] random_values
    cdef np.ndarray randoms

    x = PCG64()
    capsule = x.capsule
    if not PyCapsule_IsValid(capsule, capsule_name):
        raise ValueError("Invalid pointer to anon_func_state")
    rng = <bitgen_t *> PyCapsule_GetPointer(capsule, capsule_name)
    random_values = np.empty(n)
    # Best practice is to acquire the lock whenever generating random values.
    # This prevents other threads from modifying the state. Acquiring the lock
    # is only necessary if if the GIL is also released, as in this example.
    with x.lock, nogil:
        for i in range(n):
            random_values[i] = rng.next_double(rng.state)
    randoms = np.asarray(random_values)
    return randoms.mean()


# This function is declared nogil so it can be used without the GIL below
cdef uint32_t bounded_uint(uint32_t lb, uint32_t ub, bitgen_t *rng) nogil:
    cdef uint32_t mask, delta, val
    mask = delta = ub - lb
    mask |= mask >> 1
    mask |= mask >> 2
    mask |= mask >> 4
    mask |= mask >> 8
    mask |= mask >> 16

    val = rng.next_uint32(rng.state) & mask
    while val > delta:
        val = rng.next_uint32(rng.state) & mask
```

(continues on next page)

(continued from previous page)

```
    return lb + val


@cython.boundscheck(False)
@cython.wraparound(False)
def bounded_uints(uint32_t lb, uint32_t ub, Py_ssize_t n):
    cdef Py_ssize_t i
    cdef bitgen_t *rng
    cdef uint32_t[::1] out
    cdef const char *capsule_name = "BitGenerator"

    x = PCG64()
    out = np.empty(n, dtype=np.uint32)
    capsule = x.capsule

    if not PyCapsule_IsValid(capsule, capsule_name):
        raise ValueError("Invalid pointer to anon_func_state")
    rng = <bitgen_t *>PyCapsule_GetPointer(capsule, capsule_name)

    with x.lock, nogil:
        for i in range(n):
            out[i] = bounded_uint(lb, ub, rng)
    return np.asarray(out)
```

### extending_distributions.pyx

```
#!/usr/bin/env python3
#cython: language_level=3
"""
This file shows how the to use a BitGenerator to create a distribution.
"""
import numpy as np
cimport numpy as np
cimport cython
from cpython.pycapsule cimport PyCapsule_IsValid, PyCapsule_GetPointer
from libc.stdint cimport uint16_t, uint64_t
from numpy.random cimport bitgen_t
from numpy.random import PCG64
from numpy.random.c_distributions cimport (
      random_standard_uniform_fill, random_standard_uniform_fill_f)


@cython.boundscheck(False)
@cython.wraparound(False)
def uniforms(Py_ssize_t n):
    """
    Create an array of `n` uniformly distributed doubles.
    A 'real' distribution would want to process the values into
    some non-uniform distribution
    """
    cdef Py_ssize_t i
    cdef bitgen_t *rng
    cdef const char *capsule_name = "BitGenerator"
    cdef double[::1] random_values
```

(continues on next page)

(continued from previous page)

```
    x = PCG64()
    capsule = x.capsule
    # Optional check that the capsule if from a BitGenerator
    if not PyCapsule_IsValid(capsule, capsule_name):
        raise ValueError("Invalid pointer to anon_func_state")
    # Cast the pointer
    rng = <bitgen_t *> PyCapsule_GetPointer(capsule, capsule_name)
    random_values = np.empty(n, dtype='float64')
    with x.lock, nogil:
        for i in range(n):
            # Call the function
            random_values[i] = rng.next_double(rng.state)
    randoms = np.asarray(random_values)

    return randoms

# cython example 2
@cython.boundscheck(False)
@cython.wraparound(False)
def uint10_uniforms(Py_ssize_t n):
    """Uniform 10 bit integers stored as 16-bit unsigned integers"""
    cdef Py_ssize_t i
    cdef bitgen_t *rng
    cdef const char *capsule_name = "BitGenerator"
    cdef uint16_t[::1] random_values
    cdef int bits_remaining
    cdef int width = 10
    cdef uint64_t buff, mask = 0x3FF

    x = PCG64()
    capsule = x.capsule
    if not PyCapsule_IsValid(capsule, capsule_name):
        raise ValueError("Invalid pointer to anon_func_state")
    rng = <bitgen_t *> PyCapsule_GetPointer(capsule, capsule_name)
    random_values = np.empty(n, dtype='uint16')
    # Best practice is to release GIL and acquire the lock
    bits_remaining = 0
    with x.lock, nogil:
        for i in range(n):
            if bits_remaining < width:
                buff = rng.next_uint64(rng.state)
            random_values[i] = buff & mask
            buff >>= width

    randoms = np.asarray(random_values)
    return randoms

# cython example 3
def uniforms_ex(bit_generator, Py_ssize_t n, dtype=np.float64):
    """
    Create an array of `n` uniformly distributed doubles via a "fill" function.

    A 'real' distribution would want to process the values into
    some non-uniform distribution

    Parameters
```

(continues on next page)

(continued from previous page)

```
    ----------
    bit_generator: BitGenerator instance
    n: int
        Output vector length
    dtype: {str, dtype}, optional
        Desired dtype, either 'd' (or 'float64') or 'f' (or 'float32'). The
        default dtype value is 'd'
    """
    cdef Py_ssize_t i
    cdef bitgen_t *rng
    cdef const char *capsule_name = "BitGenerator"
    cdef np.ndarray randoms

    capsule = bit_generator.capsule
    # Optional check that the capsule if from a BitGenerator
    if not PyCapsule_IsValid(capsule, capsule_name):
        raise ValueError("Invalid pointer to anon_func_state")
    # Cast the pointer
    rng = <bitgen_t *> PyCapsule_GetPointer(capsule, capsule_name)

    _dtype = np.dtype(dtype)
    randoms = np.empty(n, dtype=_dtype)
    if _dtype == np.float32:
        with bit_generator.lock:
            random_standard_uniform_fill_f(rng, n, <float*>np.PyArray_DATA(randoms))
    elif _dtype == np.float64:
        with bit_generator.lock:
            random_standard_uniform_fill(rng, n, <double*>np.PyArray_DATA(randoms))
    else:
        raise TypeError('Unsupported dtype %r for random' % _dtype)
    return randoms
```

### Extending via CFFI

```
"""
Use cffi to access any of the underlying C functions from distributions.h
"""
import os
import numpy as np
import cffi
from .parse import parse_distributions_h
ffi = cffi.FFI()

inc_dir = os.path.join(np.get_include(), 'numpy')

# Basic numpy types
ffi.cdef('''
    typedef intptr_t npy_intp;
    typedef unsigned char npy_bool;

''')

parse_distributions_h(ffi, inc_dir)
```

(continues on next page)

(continued from previous page)

```
lib = ffi.dlopen(np.random._generator.__file__)

# Compare the distributions.h random_standard_normal_fill to
# Generator.standard_random
bit_gen = np.random.PCG64()
rng = np.random.Generator(bit_gen)
state = bit_gen.state

interface = rng.bit_generator.cffi
n = 100
vals_cffi = ffi.new('double[%d]' % n)
lib.random_standard_normal_fill(interface.bit_generator, n, vals_cffi)

# reset the state
bit_gen.state = state

vals = rng.standard_normal(n)

for i in range(n):
    assert vals[i] == vals_cffi[i]
```

### Original Source of the Generator and BitGenerators

This package was developed independently of NumPy and was integrated in version 1.17.0. The original repo is at https://github.com/bashtage/randomgen.

# 4.25 Set routines

## 4.25.1 Making proper sets

| | |
|---|---|
| `unique`(ar[, return_index, return_inverse, …]) | Find the unique elements of an array. |

## 4.25.2 Boolean operations

| | |
|---|---|
| `in1d`(ar1, ar2[, assume_unique, invert]) | Test whether each element of a 1-D array is also present in a second array. |
| `intersect1d`(ar1, ar2[, assume_unique, …]) | Find the intersection of two arrays. |
| `isin`(element, test_elements[, …]) | Calculates *element in test_elements*, broadcasting over *element* only. |
| `setdiff1d`(ar1, ar2[, assume_unique]) | Find the set difference of two arrays. |
| `setxor1d`(ar1, ar2[, assume_unique]) | Find the set exclusive-or of two arrays. |
| `union1d`(ar1, ar2) | Find the union of two arrays. |

`numpy.`**`in1d`**(*ar1*, *ar2*, *assume_unique=False*, *invert=False*)

Test whether each element of a 1-D array is also present in a second array.

Returns a boolean array the same length as *ar1* that is True where an element of *ar1* is in *ar2* and False otherwise.

We recommend using `isin` instead of `in1d` for new code.

**Parameters**

**ar1**

[(M,) array_like] Input array.

**ar2**

[array_like] The values against which to test each value of *ar1*.

**assume_unique**

[bool, optional] If True, the input arrays are both assumed to be unique, which can speed up the calculation. Default is False.

**invert**

[bool, optional] If True, the values in the returned array are inverted (that is, False where an element of *ar1* is in *ar2* and True otherwise). Default is False. `np.in1d(a, b, invert=True)` is equivalent to (but is faster than) `np.invert(in1d(a, b))`.

New in version 1.8.0.

**Returns**

**in1d**

[(M,) ndarray, bool] The values *ar1[in1d]* are in *ar2*.

**See also:**

*`isin`*

Version of this function that preserves the shape of ar1.

**`numpy.lib.arraysetops`**

Module with a number of other functions for performing set operations on arrays.

## Notes

*`in1d`* can be considered as an element-wise function version of the python keyword *in*, for 1-D sequences. `in1d(a, b)` is roughly equivalent to `np.array([item in b for item in a])`. However, this idea fails if *ar2* is a set, or similar (non-sequence) container: As `ar2` is converted to an array, in those cases `asarray(ar2)` is an object array rather than the expected array of contained values.

New in version 1.4.0.

## Examples

```
>>> test = np.array([0, 1, 2, 5, 0])
>>> states = [0, 2]
>>> mask = np.in1d(test, states)
>>> mask
array([ True, False,  True, False,  True])
>>> test[mask]
array([0, 2, 0])
>>> mask = np.in1d(test, states, invert=True)
```

(continues on next page)

(continued from previous page)

```
>>> mask
array([False,  True, False,  True, False])
>>> test[mask]
array([1, 5])
```

`numpy.`**`intersect1d`**(*ar1*, *ar2*, *assume_unique=False*, *return_indices=False*)

Find the intersection of two arrays.

Return the sorted, unique values that are in both of the input arrays.

**Parameters**

**ar1, ar2**

[array_like] Input arrays. Will be flattened if not already 1D.

**assume_unique**

[bool] If True, the input arrays are both assumed to be unique, which can speed up the calculation. Default is False.

**return_indices**

[bool] If True, the indices which correspond to the intersection of the two arrays are returned. The first instance of a value is used if there are multiple. Default is False.

New in version 1.15.0.

**Returns**

**intersect1d**

[ndarray] Sorted 1D array of common and unique elements.

**comm1**

[ndarray] The indices of the first occurrences of the common values in *ar1*. Only provided if *return_indices* is True.

**comm2**

[ndarray] The indices of the first occurrences of the common values in *ar2*. Only provided if *return_indices* is True.

**See also:**

**`numpy.lib.arraysetops`**

Module with a number of other functions for performing set operations on arrays.

### Examples

```
>>> np.intersect1d([1, 3, 4, 3], [3, 1, 2, 1])
array([1, 3])
```

To intersect more than two arrays, use functools.reduce:

```
>>> from functools import reduce
>>> reduce(np.intersect1d, ([1, 3, 4, 3], [3, 1, 2, 1], [6, 3, 4, 2]))
array([3])
```

To return the indices of the values common to the input arrays along with the intersected values:

```
>>> x = np.array([1, 1, 2, 3, 4])
>>> y = np.array([2, 1, 4, 6])
>>> xy, x_ind, y_ind = np.intersect1d(x, y, return_indices=True)
>>> x_ind, y_ind
(array([0, 2, 4]), array([1, 0, 2]))
>>> xy, x[x_ind], y[y_ind]
(array([1, 2, 4]), array([1, 2, 4]), array([1, 2, 4]))
```

`numpy.`**`isin`**(*element*, *test_elements*, *assume_unique=False*, *invert=False*)

Calculates *element in test_elements*, broadcasting over *element* only. Returns a boolean array of the same shape as *element* that is True where an element of *element* is in *test_elements* and False otherwise.

**Parameters**

**element**

[array_like] Input array.

**test_elements**

[array_like] The values against which to test each value of *element*. This argument is flattened if it is an array or array_like. See notes for behavior with non-array-like parameters.

**assume_unique**

[bool, optional] If True, the input arrays are both assumed to be unique, which can speed up the calculation. Default is False.

**invert**

[bool, optional] If True, the values in the returned array are inverted, as if calculating *element not in test_elements*. Default is False. `np.isin(a, b, invert=True)` is equivalent to (but faster than) `np.invert(np.isin(a, b))`.

**Returns**

**isin**

[ndarray, bool] Has the same shape as *element*. The values *element[isin]* are in *test_elements*.

**See also:**

***`in1d`***

Flattened version of this function.

**numpy.lib.arraysetops**

Module with a number of other functions for performing set operations on arrays.

### Notes

*isin* is an element-wise function version of the python keyword *in*. `isin(a, b)` is roughly equivalent to `np.array([item in b for item in a])` if *a* and *b* are 1-D sequences.

*element* and *test_elements* are converted to arrays if they are not already. If *test_elements* is a set (or other non-sequence collection) it will be converted to an object array with one element, rather than an array of the values contained in *test_elements*. This is a consequence of the *array* constructor's way of handling non-sequence collections. Converting the set to a list usually gives the desired behavior.

New in version 1.13.0.

### Examples

```
>>> element = 2*np.arange(4).reshape((2, 2))
>>> element
array([[0, 2],
       [4, 6]])
>>> test_elements = [1, 2, 4, 8]
>>> mask = np.isin(element, test_elements)
>>> mask
array([[False,  True],
       [ True, False]])
>>> element[mask]
array([2, 4])
```

The indices of the matched values can be obtained with *nonzero*:

```
>>> np.nonzero(mask)
(array([0, 1]), array([1, 0]))
```

The test can also be inverted:

```
>>> mask = np.isin(element, test_elements, invert=True)
>>> mask
array([[ True, False],
       [False,  True]])
>>> element[mask]
array([0, 6])
```

Because of how *array* handles sets, the following does not work as expected:

```
>>> test_set = {1, 2, 4, 8}
>>> np.isin(element, test_set)
array([[False, False],
       [False, False]])
```

Casting the set to a list gives the expected result:

```
>>> np.isin(element, list(test_set))
array([[False,  True],
       [ True, False]])
```

numpy.**setdiff1d**(*ar1*, *ar2*, *assume_unique=False*)

Find the set difference of two arrays.

Return the unique values in *ar1* that are not in *ar2*.

**Parameters**

**ar1**

[array_like] Input array.

**ar2**

[array_like] Input comparison array.

**assume_unique**

[bool] If True, the input arrays are both assumed to be unique, which can speed up the calculation. Default is False.

**Returns**

**setdiff1d**

[ndarray] 1D array of values in *ar1* that are not in *ar2*. The result is sorted when *assume_unique=False*, but otherwise only sorted if the input is sorted.

**See also:**

**`numpy.lib.arraysetops`**

Module with a number of other functions for performing set operations on arrays.

### Examples

```
>>> a = np.array([1, 2, 3, 2, 4, 1])
>>> b = np.array([3, 4, 5, 6])
>>> np.setdiff1d(a, b)
array([1, 2])
```

numpy.**setxor1d**(*ar1*, *ar2*, *assume_unique=False*)

Find the set exclusive-or of two arrays.

Return the sorted, unique values that are in only one (not both) of the input arrays.

**Parameters**

**ar1, ar2**

[array_like] Input arrays.

**assume_unique**

[bool] If True, the input arrays are both assumed to be unique, which can speed up the calculation. Default is False.

**Returns**

**setxor1d**

[ndarray] Sorted 1D array of unique values that are in only one of the input arrays.

### Examples

```
>>> a = np.array([1, 2, 3, 2, 4])
>>> b = np.array([2, 3, 5, 7, 5])
>>> np.setxor1d(a,b)
array([1, 4, 5, 7])
```

`numpy.`**`union1d`**`(`*ar1*, *ar2*`)`

Find the union of two arrays.

Return the unique, sorted array of values that are in either of the two input arrays.

**Parameters**

**ar1, ar2**

[array_like] Input arrays. They are flattened if they are not already 1D.

**Returns**

**union1d**

[ndarray] Unique, sorted union of the input arrays.

**See also:**

**`numpy.lib.arraysetops`**

Module with a number of other functions for performing set operations on arrays.

### Examples

```
>>> np.union1d([-1, 0, 1], [-2, 0, 2])
array([-2, -1,  0,  1,  2])
```

To find the union of more than two arrays, use functools.reduce:

```
>>> from functools import reduce
>>> reduce(np.union1d, ([1, 3, 4, 3], [3, 1, 2, 1], [6, 3, 4, 2]))
array([1, 2, 3, 4, 6])
```

## 4.26 Sorting, searching, and counting

### 4.26.1 Sorting

| | |
|---|---|
| `sort`(a[, axis, kind, order]) | Return a sorted copy of an array. |
| `lexsort`(keys[, axis]) | Perform an indirect stable sort using a sequence of keys. |
| `argsort`(a[, axis, kind, order]) | Returns the indices that would sort an array. |
| `ndarray.sort`([axis, kind, order]) | Sort an array in-place. |
| `msort`(a) | Return a copy of an array sorted along the first axis. |

Continued on next page

Table 169 – continued from previous page

| | |
|---|---|
| `sort_complex`(a) | Sort a complex array using the real part first, then the imaginary part. |
| `partition`(a, kth[, axis, kind, order]) | Return a partitioned copy of an array. |
| `argpartition`(a, kth[, axis, kind, order]) | Perform an indirect partition along the given axis using the algorithm specified by the *kind* keyword. |

`numpy.`**`sort`**(*a*, *axis=-1*, *kind=None*, *order=None*)

Return a sorted copy of an array.

**Parameters**

**a**

[array_like] Array to be sorted.

**axis**

[int or None, optional] Axis along which to sort. If None, the array is flattened before sorting. The default is -1, which sorts along the last axis.

**kind**

[{'quicksort', 'mergesort', 'heapsort', 'stable'}, optional] Sorting algorithm. The default is 'quicksort'. Note that both 'stable' and 'mergesort' use timsort or radix sort under the covers and, in general, the actual implementation will vary with data type. The 'mergesort' option is retained for backwards compatibility.

Changed in version 1.15.0.: The 'stable' option was added.

**order**

[str or list of str, optional] When *a* is an array with fields defined, this argument specifies which fields to compare first, second, etc. A single field can be specified as a string, and not all fields need be specified, but unspecified fields will still be used, in the order in which they come up in the dtype, to break ties.

**Returns**

**sorted_array**

[ndarray] Array of the same type and shape as *a*.

**See also:**

**`ndarray.sort`**

Method to sort an array in-place.

**`argsort`**

Indirect sort.

**`lexsort`**

Indirect stable sort on multiple keys.

**`searchsorted`**

Find elements in a sorted array.

`partition`

Partial sort.

## Notes

The various sorting algorithms are characterized by their average speed, worst case performance, work space size, and whether they are stable. A stable sort keeps items with the same key in the same relative order. The four algorithms implemented in NumPy have the following properties:

| kind | speed | worst case | work space | stable |
|---|---|---|---|---|
| 'quicksort' | 1 | O(n^2) | 0 | no |
| 'heapsort' | 3 | O(n*log(n)) | 0 | no |
| 'mergesort' | 2 | O(n*log(n)) | ~n/2 | yes |
| 'timsort' | 2 | O(n*log(n)) | ~n/2 | yes |

**Note:** The datatype determines which of 'mergesort' or 'timsort' is actually used, even if 'mergesort' is specified. User selection at a finer scale is not currently available.

All the sort algorithms make temporary copies of the data when sorting along any but the last axis. Consequently, sorting along the last axis is faster and uses less space than sorting along any other axis.

The sort order for complex numbers is lexicographic. If both the real and imaginary parts are non-nan then the order is determined by the real parts except when they are equal, in which case the order is determined by the imaginary parts.

Previous to numpy 1.4.0 sorting real and complex arrays containing nan values led to undefined behaviour. In numpy versions >= 1.4.0 nan values are sorted to the end. The extended sort order is:

- Real: [R, nan]
- Complex: [R + Rj, R + nanj, nan + Rj, nan + nanj]

where R is a non-nan real value. Complex values with the same nan placements are sorted according to the non-nan part if it exists. Non-nan values are sorted as before.

New in version 1.12.0.

quicksort has been changed to introsort. When sorting does not make enough progress it switches to heapsort. This implementation makes quicksort O(n*log(n)) in the worst case.

'stable' automatically chooses the best stable sorting algorithm for the data type being sorted. It, along with 'mergesort' is currently mapped to timsort or radix sort depending on the data type. API forward compatibility currently limits the ability to select the implementation and it is hardwired for the different data types.

New in version 1.17.0.

Timsort is added for better performance on already or nearly sorted data. On random data timsort is almost identical to mergesort. It is now used for stable sort while quicksort is still the default sort if none is chosen. For timsort details, refer to CPython listsort.txt. 'mergesort' and 'stable' are mapped to radix sort for integer data types. Radix sort is an O(n) sort instead of O(n log n).

Changed in version 1.18.0.

NaT now sorts to the end of arrays for consistency with NaN.

### Examples

```
>>> a = np.array([[1,4],[3,1]])
>>> np.sort(a)                # sort along the last axis
array([[1, 4],
       [1, 3]])
>>> np.sort(a, axis=None)     # sort the flattened array
array([1, 1, 3, 4])
>>> np.sort(a, axis=0)        # sort along the first axis
array([[1, 1],
       [3, 4]])
```

Use the *order* keyword to specify a field to use when sorting a structured array:

```
>>> dtype = [('name', 'S10'), ('height', float), ('age', int)]
>>> values = [('Arthur', 1.8, 41), ('Lancelot', 1.9, 38),
...           ('Galahad', 1.7, 38)]
>>> a = np.array(values, dtype=dtype)       # create a structured array
>>> np.sort(a, order='height')
array([('Galahad', 1.7, 38), ('Arthur', 1.8, 41),
       ('Lancelot', 1.8999999999999999, 38)],
      dtype=[('name', '|S10'), ('height', '<f8'), ('age', '<i4')])
```

Sort by age, then height if ages are equal:

```
>>> np.sort(a, order=['age', 'height'])
array([('Galahad', 1.7, 38), ('Lancelot', 1.8999999999999999, 38),
       ('Arthur', 1.8, 41)],
      dtype=[('name', '|S10'), ('height', '<f8'), ('age', '<i4')])
```

`numpy.`**`lexsort`**(*keys*, *axis=-1*)

Perform an indirect stable sort using a sequence of keys.

Given multiple sorting keys, which can be interpreted as columns in a spreadsheet, lexsort returns an array of integer indices that describes the sort order by multiple columns. The last key in the sequence is used for the primary sort order, the second-to-last key for the secondary sort order, and so on. The keys argument must be a sequence of objects that can be converted to arrays of the same shape. If a 2D array is provided for the keys argument, it's rows are interpreted as the sorting keys and sorting is according to the last row, second last row etc.

**Parameters**

**keys**

[(k, N) array or tuple containing k (N,)-shaped sequences] The *k* different "columns" to be sorted. The last column (or row if *keys* is a 2D array) is the primary sort key.

**axis**

[int, optional] Axis to be indirectly sorted. By default, sort over the last axis.

**Returns**

**indices**

[(N,) ndarray of ints] Array of indices that sort the keys along the specified axis.

**See also:**

***argsort***

Indirect sort.

***ndarray.sort***

In-place sort.

***sort***

Return a sorted copy of an array.

### Examples

Sort names: first by surname, then by name.

```
>>> surnames =    ('Hertz',    'Galilei', 'Hertz')
>>> first_names = ('Heinrich', 'Galileo', 'Gustav')
>>> ind = np.lexsort((first_names, surnames))
>>> ind
array([1, 2, 0])
```

```
>>> [surnames[i] + ", " + first_names[i] for i in ind]
['Galilei, Galileo', 'Hertz, Gustav', 'Hertz, Heinrich']
```

Sort two columns of numbers:

```
>>> a = [1,5,1,4,3,4,4] # First column
>>> b = [9,4,0,4,0,2,1] # Second column
>>> ind = np.lexsort((b,a)) # Sort by a, then by b
>>> ind
array([2, 0, 4, 6, 5, 3, 1])
```

```
>>> [(a[i],b[i]) for i in ind]
[(1, 0), (1, 9), (3, 0), (4, 1), (4, 2), (4, 4), (5, 4)]
```

Note that sorting is first according to the elements of `a`. Secondary sorting is according to the elements of `b`.

A normal `argsort` would have yielded:

```
>>> [(a[i],b[i]) for i in np.argsort(a)]
[(1, 9), (1, 0), (3, 0), (4, 4), (4, 2), (4, 1), (5, 4)]
```

Structured arrays are sorted lexically by `argsort`:

```
>>> x = np.array([(1,9), (5,4), (1,0), (4,4), (3,0), (4,2), (4,1)],
...              dtype=np.dtype([('x', int), ('y', int)]))
```

```
>>> np.argsort(x) # or np.argsort(x, order=('x', 'y'))
array([2, 0, 4, 6, 5, 3, 1])
```

`numpy.`**`argsort`**(*a*, *axis=-1*, *kind=None*, *order=None*)

Returns the indices that would sort an array.

Perform an indirect sort along the given axis using the algorithm specified by the *kind* keyword. It returns an array of indices of the same shape as *a* that index data along the given axis in sorted order.

**Parameters**

**a**

[array_like] Array to sort.

**axis**

[int or None, optional] Axis along which to sort. The default is -1 (the last axis). If None, the flattened array is used.

**kind**

[{'quicksort', 'mergesort', 'heapsort', 'stable'}, optional] Sorting algorithm. The default is 'quicksort'. Note that both 'stable' and 'mergesort' use timsort under the covers and, in general, the actual implementation will vary with data type. The 'mergesort' option is retained for backwards compatibility.

Changed in version 1.15.0.: The 'stable' option was added.

**order**

[str or list of str, optional] When *a* is an array with fields defined, this argument specifies which fields to compare first, second, etc. A single field can be specified as a string, and not all fields need be specified, but unspecified fields will still be used, in the order in which they come up in the dtype, to break ties.

**Returns**

**index_array**

[ndarray, int] Array of indices that sort *a* along the specified *axis*. If *a* is one-dimensional, `a[index_array]` yields a sorted *a*. More generally, `np.take_along_axis(a, index_array, axis=axis)` always yields the sorted *a*, irrespective of dimensionality.

**See also:**

***`sort`***

Describes sorting algorithms used.

***`lexsort`***

Indirect stable sort with multiple keys.

***`ndarray.sort`***

Inplace sort.

***`argpartition`***

Indirect partial sort.

***`take_along_axis`***

Apply `index_array` from argsort to an array as if by calling sort.

### Notes

See `sort` for notes on the different sorting algorithms.

As of NumPy 1.4.0 `argsort` works with real/complex arrays containing nan values. The enhanced sort order is documented in `sort`.

### Examples

One dimensional array:

```
>>> x = np.array([3, 1, 2])
>>> np.argsort(x)
array([1, 2, 0])
```

Two-dimensional array:

```
>>> x = np.array([[0, 3], [2, 2]])
>>> x
array([[0, 3],
       [2, 2]])
```

```
>>> ind = np.argsort(x, axis=0)  # sorts along first axis (down)
>>> ind
array([[0, 1],
       [1, 0]])
>>> np.take_along_axis(x, ind, axis=0)  # same as np.sort(x, axis=0)
array([[0, 2],
       [2, 3]])
```

```
>>> ind = np.argsort(x, axis=1)  # sorts along last axis (across)
>>> ind
array([[0, 1],
       [0, 1]])
>>> np.take_along_axis(x, ind, axis=1)  # same as np.sort(x, axis=1)
array([[0, 3],
       [2, 2]])
```

Indices of the sorted elements of a N-dimensional array:

```
>>> ind = np.unravel_index(np.argsort(x, axis=None), x.shape)
>>> ind
(array([0, 1, 1, 0]), array([0, 0, 1, 1]))
>>> x[ind]  # same as np.sort(x, axis=None)
array([0, 2, 2, 3])
```

Sorting with keys:

```
>>> x = np.array([(1, 0), (0, 1)], dtype=[('x', '<i4'), ('y', '<i4')])
>>> x
array([(1, 0), (0, 1)],
      dtype=[('x', '<i4'), ('y', '<i4')])
```

```
>>> np.argsort(x, order=('x','y'))
array([1, 0])
```

```
>>> np.argsort(x, order=('y','x'))
array([0, 1])
```

`numpy.`**`msort`**`(`*a*`)`

Return a copy of an array sorted along the first axis.

**Parameters**

**a**

[array_like] Array to be sorted.

**Returns**

**sorted_array**

[ndarray] Array of the same type and shape as *a*.

**See also:**

*`sort`*

## Notes

`np.msort(a)` is equivalent to `np.sort(a, axis=0)`.

`numpy.`**`sort_complex`**`(`*a*`)`

Sort a complex array using the real part first, then the imaginary part.

**Parameters**

**a**

[array_like] Input array

**Returns**

**out**

[complex ndarray] Always returns a sorted complex array.

## Examples

```
>>> np.sort_complex([5, 3, 6, 2, 1])
array([1.+0.j, 2.+0.j, 3.+0.j, 5.+0.j, 6.+0.j])
```

```
>>> np.sort_complex([1 + 2j, 2 - 1j, 3 - 2j, 3 - 3j, 3 + 5j])
array([1.+2.j,  2.-1.j,  3.-3.j,  3.-2.j,  3.+5.j])
```

`numpy.`**`partition`**`(`*a*, *kth*, *axis=-1*, *kind='introselect'*, *order=None*`)`

Return a partitioned copy of an array.

Creates a copy of the array with its elements rearranged in such a way that the value of the element in k-th position is in the position it would be in a sorted array. All elements smaller than the k-th element are moved before this element and all equal or greater are moved behind it. The ordering of the elements in the two partitions is undefined.

New in version 1.8.0.

**Parameters**

**a**

[array_like] Array to be sorted.

**kth**

[int or sequence of ints] Element index to partition by. The k-th value of the element will be in its final sorted position and all smaller elements will be moved before it and all equal or greater elements behind it. The order of all elements in the partitions is undefined. If provided with a sequence of k-th it will partition all elements indexed by k-th of them into their sorted position at once.

**axis**

[int or None, optional] Axis along which to sort. If None, the array is flattened before sorting. The default is -1, which sorts along the last axis.

**kind**

[{'introselect'}, optional] Selection algorithm. Default is 'introselect'.

**order**

[str or list of str, optional] When *a* is an array with fields defined, this argument specifies which fields to compare first, second, etc. A single field can be specified as a string. Not all fields need be specified, but unspecified fields will still be used, in the order in which they come up in the dtype, to break ties.

**Returns**

**partitioned_array**

[ndarray] Array of the same type and shape as *a*.

**See also:**

`ndarray.partition`

Method to sort an array in-place.

`argpartition`

Indirect partition.

`sort`

Full sorting

## Notes

The various selection algorithms are characterized by their average speed, worst case performance, work space size, and whether they are stable. A stable sort keeps items with the same key in the same relative order. The available algorithms have the following properties:

| kind | speed | worst case | work space | stable |
|---|---|---|---|---|
| 'introselect' | 1 | O(n) | 0 | no |

All the partition algorithms make temporary copies of the data when partitioning along any but the last axis. Consequently, partitioning along the last axis is faster and uses less space than partitioning along any other axis.

The sort order for complex numbers is lexicographic. If both the real and imaginary parts are non-nan then the order is determined by the real parts except when they are equal, in which case the order is determined by the imaginary parts.

### Examples

```
>>> a = np.array([3, 4, 2, 1])
>>> np.partition(a, 3)
array([2, 1, 3, 4])
```

```
>>> np.partition(a, (1, 3))
array([1, 2, 3, 4])
```

`numpy.`**`argpartition`**`(`*a*, *kth*, *axis=-1*, *kind='introselect'*, *order=None*`)`

Perform an indirect partition along the given axis using the algorithm specified by the *kind* keyword. It returns an array of indices of the same shape as *a* that index data along the given axis in partitioned order.

New in version 1.8.0.

**Parameters**

**a**

[array_like] Array to sort.

**kth**

[int or sequence of ints] Element index to partition by. The k-th element will be in its final sorted position and all smaller elements will be moved before it and all larger elements behind it. The order all elements in the partitions is undefined. If provided with a sequence of k-th it will partition all of them into their sorted position at once.

**axis**

[int or None, optional] Axis along which to sort. The default is -1 (the last axis). If None, the flattened array is used.

**kind**

[{'introselect'}, optional] Selection algorithm. Default is 'introselect'

**order**

[str or list of str, optional] When *a* is an array with fields defined, this argument specifies which fields to compare first, second, etc. A single field can be specified as a string, and not all fields need be specified, but unspecified fields will still be used, in the order in which they come up in the dtype, to break ties.

**Returns**

**index_array**

[ndarray, int] Array of indices that partition *a* along the specified axis. If *a* is one-dimensional, `a[index_array]` yields a partitioned *a*. More generally, `np.take_along_axis(a, index_array, axis=a)` always yields the partitioned *a*, irrespective of dimensionality.

**See also:**

***partition***

Describes partition algorithms used.

***ndarray.partition***

Inplace partition.

***argsort***

Full indirect sort.

***take_along_axis***

Apply `index_array` from argpartition to an array as if by calling partition.

### Notes

See *partition* for notes on the different selection algorithms.

### Examples

One dimensional array:

```
>>> x = np.array([3, 4, 2, 1])
>>> x[np.argpartition(x, 3)]
array([2, 1, 3, 4])
>>> x[np.argpartition(x, (1, 3))]
array([1, 2, 3, 4])
```

```
>>> x = [3, 4, 2, 1]
>>> np.array(x)[np.argpartition(x, 3)]
array([2, 1, 3, 4])
```

Multi-dimensional array:

```
>>> x = np.array([[3, 4, 2], [1, 3, 1]])
>>> index_array = np.argpartition(x, kth=1, axis=-1)
>>> np.take_along_axis(x, index_array, axis=-1)  # same as np.partition(x, kth=1)
array([[2, 3, 4],
       [1, 1, 3]])
```

## 4.26.2 Searching

| | |
|---|---|
| *argmax*(a[, axis, out]) | Returns the indices of the maximum values along an axis. |
| *nanargmax*(a[, axis]) | Return the indices of the maximum values in the specified axis ignoring NaNs. |
| *argmin*(a[, axis, out]) | Returns the indices of the minimum values along an axis. |
| *nanargmin*(a[, axis]) | Return the indices of the minimum values in the specified axis ignoring NaNs. |
| *argwhere*(a) | Find the indices of array elements that are non-zero, grouped by element. |

Continued on next page

Table 170 – continued from previous page

| | |
|---|---|
| `nonzero`(a) | Return the indices of the elements that are non-zero. |
| `flatnonzero`(a) | Return indices that are non-zero in the flattened version of a. |
| `where`(condition, [x, y]) | Return elements chosen from *x* or *y* depending on *condition*. |
| `searchsorted`(a, v[, side, sorter]) | Find indices where elements should be inserted to maintain order. |
| `extract`(condition, arr) | Return the elements of an array that satisfy some condition. |

`numpy.`**`argmax`**(*a*, *axis=None*, *out=None*)

Returns the indices of the maximum values along an axis.

**Parameters**

**a**

[array_like] Input array.

**axis**

[int, optional] By default, the index is into the flattened array, otherwise along the specified axis.

**out**

[array, optional] If provided, the result will be inserted into this array. It should be of the appropriate shape and dtype.

**Returns**

**index_array**

[ndarray of ints] Array of indices into the array. It has the same shape as *a.shape* with the dimension along *axis* removed.

**See also:**

`ndarray.argmax`, `argmin`

**`amax`**

The maximum value along a given axis.

**`unravel_index`**

Convert a flat index into an index tuple.

**`take_along_axis`**

Apply `np.expand_dims(index_array, axis)` from argmax to an array as if by calling max.

### Notes

In case of multiple occurrences of the maximum values, the indices corresponding to the first occurrence are returned.

### Examples

```
>>> a = np.arange(6).reshape(2,3) + 10
>>> a
array([[10, 11, 12],
       [13, 14, 15]])
>>> np.argmax(a)
5
>>> np.argmax(a, axis=0)
array([1, 1, 1])
>>> np.argmax(a, axis=1)
array([2, 2])
```

Indexes of the maximal elements of a N-dimensional array:

```
>>> ind = np.unravel_index(np.argmax(a, axis=None), a.shape)
>>> ind
(1, 2)
>>> a[ind]
15
```

```
>>> b = np.arange(6)
>>> b[1] = 5
>>> b
array([0, 5, 2, 3, 4, 5])
>>> np.argmax(b)  # Only the first occurrence is returned.
1
```

```
>>> x = np.array([[4,2,3], [1,0,3]])
>>> index_array = np.argmax(x, axis=-1)
>>> # Same as np.max(x, axis=-1, keepdims=True)
>>> np.take_along_axis(x, np.expand_dims(index_array, axis=-1), axis=-1)
array([[4],
       [3]])
>>> # Same as np.max(x, axis=-1)
>>> np.take_along_axis(x, np.expand_dims(index_array, axis=-1), axis=-1).
↪squeeze(axis=-1)
array([4, 3])
```

`numpy.`**`nanargmax`**`(`*a*, *axis=None*`)`

Return the indices of the maximum values in the specified axis ignoring NaNs. For all-NaN slices `ValueError` is raised. Warning: the results cannot be trusted if a slice contains only NaNs and -Infs.

**Parameters**

**a**

[array_like] Input data.

**axis**

[int, optional] Axis along which to operate. By default flattened input is used.

**Returns**

**index_array**

[ndarray] An array of indices or a single index value.

**See also:**

*argmax*, *nanargmin*

## Examples

```
>>> a = np.array([[np.nan, 4], [2, 3]])
>>> np.argmax(a)
0
>>> np.nanargmax(a)
1
>>> np.nanargmax(a, axis=0)
array([1, 0])
>>> np.nanargmax(a, axis=1)
array([1, 1])
```

`numpy.`**`argmin`**(*a*, *axis=None*, *out=None*)

Returns the indices of the minimum values along an axis.

**Parameters**

**a**

[array_like] Input array.

**axis**

[int, optional] By default, the index is into the flattened array, otherwise along the specified axis.

**out**

[array, optional] If provided, the result will be inserted into this array. It should be of the appropriate shape and dtype.

**Returns**

**index_array**

[ndarray of ints] Array of indices into the array. It has the same shape as *a.shape* with the dimension along *axis* removed.

**See also:**

*ndarray.argmin*, *argmax*

***amin***

The minimum value along a given axis.

***unravel_index***

Convert a flat index into an index tuple.

*take_along_axis*
Apply `np.expand_dims(index_array, axis)` from argmin to an array as if by calling min.

### Notes

In case of multiple occurrences of the minimum values, the indices corresponding to the first occurrence are returned.

### Examples

```
>>> a = np.arange(6).reshape(2,3) + 10
>>> a
array([[10, 11, 12],
       [13, 14, 15]])
>>> np.argmin(a)
0
>>> np.argmin(a, axis=0)
array([0, 0, 0])
>>> np.argmin(a, axis=1)
array([0, 0])
```

Indices of the minimum elements of a N-dimensional array:

```
>>> ind = np.unravel_index(np.argmin(a, axis=None), a.shape)
>>> ind
(0, 0)
>>> a[ind]
10
```

```
>>> b = np.arange(6) + 10
>>> b[4] = 10
>>> b
array([10, 11, 12, 13, 10, 15])
>>> np.argmin(b)  # Only the first occurrence is returned.
0
```

```
>>> x = np.array([[4,2,3], [1,0,3]])
>>> index_array = np.argmin(x, axis=-1)
>>> # Same as np.min(x, axis=-1, keepdims=True)
>>> np.take_along_axis(x, np.expand_dims(index_array, axis=-1), axis=-1)
array([[2],
       [0]])
>>> # Same as np.max(x, axis=-1)
>>> np.take_along_axis(x, np.expand_dims(index_array, axis=-1), axis=-1).
↪squeeze(axis=-1)
array([2, 0])
```

numpy.**nanargmin**(*a*, *axis=None*)
Return the indices of the minimum values in the specified axis ignoring NaNs. For all-NaN slices `ValueError` is raised. Warning: the results cannot be trusted if a slice contains only NaNs and Infs.

**Parameters**

**a**

[array_like] Input data.

**axis**

[int, optional] Axis along which to operate. By default flattened input is used.

**Returns**

**index_array**

[ndarray] An array of indices or a single index value.

**See also:**

*argmin*, *nanargmax*

### Examples

```
>>> a = np.array([[np.nan, 4], [2, 3]])
>>> np.argmin(a)
0
>>> np.nanargmin(a)
2
>>> np.nanargmin(a, axis=0)
array([1, 1])
>>> np.nanargmin(a, axis=1)
array([1, 0])
```

`numpy.`**`argwhere`**`(a)`

Find the indices of array elements that are non-zero, grouped by element.

**Parameters**

**a**

[array_like] Input data.

**Returns**

**index_array**

[(N, a.ndim) ndarray] Indices of elements that are non-zero. Indices are grouped by element. This array will have shape `(N, a.ndim)` where `N` is the number of non-zero items.

**See also:**

*where*, *nonzero*

## Notes

`np.argwhere(a)` is almost the same as `np.transpose(np.nonzero(a))`, but produces a result of the correct shape for a 0D array.

The output of `argwhere` is not suitable for indexing arrays. For this purpose use `nonzero(a)` instead.

## Examples

```
>>> x = np.arange(6).reshape(2,3)
>>> x
array([[0, 1, 2],
       [3, 4, 5]])
>>> np.argwhere(x>1)
array([[0, 2],
       [1, 0],
       [1, 1],
       [1, 2]])
```

`numpy.`**`flatnonzero`**`(`*a*`)`

Return indices that are non-zero in the flattened version of a.

This is equivalent to np.nonzero(np.ravel(a))[0].

**Parameters**

**a**

[array_like] Input data.

**Returns**

**res**

[ndarray] Output array, containing the indices of the elements of *a.ravel()* that are non-zero.

**See also:**

***`nonzero`***

Return the indices of the non-zero elements of the input array.

***`ravel`***

Return a 1-D array containing the elements of the input array.

## Examples

```
>>> x = np.arange(-2, 3)
>>> x
array([-2, -1,  0,  1,  2])
>>> np.flatnonzero(x)
array([0, 1, 3, 4])
```

Use the indices of the non-zero elements as an index array to extract these elements:

```
>>> x.ravel()[np.flatnonzero(x)]
array([-2, -1,  1,  2])
```

`numpy.`**`searchsorted`**(*a*, *v*, *side='left'*, *sorter=None*)

Find indices where elements should be inserted to maintain order.

Find the indices into a sorted array *a* such that, if the corresponding elements in *v* were inserted before the indices, the order of *a* would be preserved.

Assuming that *a* is sorted:

| *side* | returned index *i* satisfies |
|---|---|
| left | `a[i-1] < v <= a[i]` |
| right | `a[i-1] <= v < a[i]` |

**Parameters**

**a**

[1-D array_like] Input array. If *sorter* is None, then it must be sorted in ascending order, otherwise *sorter* must be an array of indices that sort it.

**v**

[array_like] Values to insert into *a*.

**side**

[{'left', 'right'}, optional] If 'left', the index of the first suitable location found is given. If 'right', return the last such index. If there is no suitable index, return either 0 or N (where N is the length of *a*).

**sorter**

[1-D array_like, optional] Optional array of integer indices that sort array a into ascending order. They are typically the result of argsort.

New in version 1.7.0.

**Returns**

**indices**

[array of ints] Array of insertion points with the same shape as *v*.

**See also:**

**`sort`**

Return a sorted copy of an array.

**`histogram`**

Produce histogram from 1-D data.

## Notes

Binary search is used to find the required insertion points.

As of NumPy 1.4.0 *`searchsorted`* works with real/complex arrays containing *`nan`* values. The enhanced sort order is documented in *`sort`*.

This function uses the same algorithm as the builtin python `bisect.bisect_left` (`side='left'`) and `bisect.bisect_right` (`side='right'`) functions, which is also vectorized in the *v* argument.

## Examples

```
>>> np.searchsorted([1,2,3,4,5], 3)
2
>>> np.searchsorted([1,2,3,4,5], 3, side='right')
3
>>> np.searchsorted([1,2,3,4,5], [-10, 10, 2, 3])
array([0, 5, 1, 2])
```

`numpy.`**`extract`**(*condition*, *arr*)

Return the elements of an array that satisfy some condition.

This is equivalent to `np.compress(ravel(condition), ravel(arr))`. If *condition* is boolean `np.extract` is equivalent to `arr[condition]`.

Note that *`place`* does the exact opposite of *`extract`*.

**Parameters**

**condition**

[array_like] An array whose nonzero or True entries indicate the elements of *arr* to extract.

**arr**

[array_like] Input array of the same size as *condition*.

**Returns**

**extract**

[ndarray] Rank 1 array of values from *arr* where *condition* is True.

**See also:**

*`take`*, *`put`*, *`copyto`*, *`compress`*, *`place`*

## Examples

```
>>> arr = np.arange(12).reshape((3, 4))
>>> arr
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
>>> condition = np.mod(arr, 3)==0
>>> condition
array([[ True, False, False,  True],
```

(continues on next page)

(continued from previous page)

```
       [False, False,  True, False],
       [False,  True, False, False]])
>>> np.extract(condition, arr)
array([0, 3, 6, 9])
```

If *condition* is boolean:

```
>>> arr[condition]
array([0, 3, 6, 9])
```

### 4.26.3 Counting

| | |
|---|---|
| *count_nonzero*(a[, axis, keepdims]) | Counts the number of non-zero values in the array `a`. |

`numpy.`**`count_nonzero`**(*a*, *axis=None*, *, *keepdims=False*)

Counts the number of non-zero values in the array `a`.

The word "non-zero" is in reference to the Python 2.x built-in method `__nonzero__()` (renamed `__bool__()` in Python 3.x) of Python objects that tests an object's "truthfulness". For example, any number is considered truthful if it is nonzero, whereas any string is considered truthful if it is not the empty string. Thus, this function (recursively) counts how many elements in `a` (and in sub-arrays thereof) have their `__nonzero__()` or `__bool__()` method evaluated to `True`.

**Parameters**

**a**

[array_like] The array for which to count non-zeros.

**axis**

[int or tuple, optional] Axis or tuple of axes along which to count non-zeros. Default is None, meaning that non-zeros will be counted along a flattened version of `a`.

New in version 1.12.0.

**keepdims**

[bool, optional] If this is set to True, the axes that are counted are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the input array.

New in version 1.19.0.

**Returns**

**count**

[int or array of int] Number of non-zero values in the array along a given axis. Otherwise, the total number of non-zero values in the array is returned.

**See also:**

***nonzero***

Return the coordinates of all the non-zero values.

### Examples

```
>>> np.count_nonzero(np.eye(4))
4
>>> a = np.array([[0, 1, 7, 0],
...               [3, 0, 2, 19]])
>>> np.count_nonzero(a)
5
>>> np.count_nonzero(a, axis=0)
array([1, 1, 2, 1])
>>> np.count_nonzero(a, axis=1)
array([2, 3])
>>> np.count_nonzero(a, axis=1, keepdims=True)
array([[2],
       [3]])
```

## 4.27 Statistics

### 4.27.1 Order statistics

| | |
|---|---|
| `amin`(a[, axis, out, keepdims, initial, where]) | Return the minimum of an array or minimum along an axis. |
| `amax`(a[, axis, out, keepdims, initial, where]) | Return the maximum of an array or maximum along an axis. |
| `nanmin`(a[, axis, out, keepdims]) | Return minimum of an array or minimum along an axis, ignoring any NaNs. |
| `nanmax`(a[, axis, out, keepdims]) | Return the maximum of an array or maximum along an axis, ignoring any NaNs. |
| `ptp`(a[, axis, out, keepdims]) | Range of values (maximum - minimum) along an axis. |
| `percentile`(a, q[, axis, out, …]) | Compute the q-th percentile of the data along the specified axis. |
| `nanpercentile`(a, q[, axis, out, …]) | Compute the qth percentile of the data along the specified axis, while ignoring nan values. |
| `quantile`(a, q[, axis, out, overwrite_input, …]) | Compute the q-th quantile of the data along the specified axis. |
| `nanquantile`(a, q[, axis, out, …]) | Compute the qth quantile of the data along the specified axis, while ignoring nan values. |

`numpy.`**`amin`**(*a*, *axis=None*, *out=None*, *keepdims=<no value>*, *initial=<no value>*, *where=<no value>*)

Return the minimum of an array or minimum along an axis.

**Parameters**

**a**

[array_like] Input data.

**axis**

[None or int or tuple of ints, optional] Axis or axes along which to operate. By default, flattened input is used.

New in version 1.7.0.

If this is a tuple of ints, the minimum is selected over multiple axes, instead of a single axis or all the axes as before.

**out**

[ndarray, optional] Alternative output array in which to place the result. Must be of the same shape and buffer length as the expected output. See *ufuncs-output-type* for more details.

**keepdims**

[bool, optional] If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the input array.

If the default value is passed, then *keepdims* will not be passed through to the `amin` method of sub-classes of `ndarray`, however any non-default value will be. If the sub-class' method does not implement *keepdims* any exceptions will be raised.

**initial**

[scalar, optional] The maximum value of an output element. Must be present to allow computation on empty slice. See `reduce` for details.

New in version 1.15.0.

**where**

[array_like of bool, optional] Elements to compare for the minimum. See `reduce` for details.

New in version 1.17.0.

**Returns**

**amin**

[ndarray or scalar] Minimum of *a*. If *axis* is None, the result is a scalar value. If *axis* is given, the result is an array of dimension `a.ndim - 1`.

**See also:**

**`amax`**

The maximum value of an array along a given axis, propagating any NaNs.

**`nanmin`**

The minimum value of an array along a given axis, ignoring any NaNs.

**`minimum`**

Element-wise minimum of two arrays, propagating any NaNs.

**`fmin`**

Element-wise minimum of two arrays, ignoring any NaNs.

**`argmin`**

Return the indices of the minimum values.

`nanmax`, `maximum`, `fmax`

### Notes

NaN values are propagated, that is if at least one item is NaN, the corresponding min value will be NaN as well. To ignore NaN values (MATLAB behavior), please use nanmin.

Don't use *amin* for element-wise comparison of 2 arrays; when `a.shape[0]` is 2, `minimum(a[0], a[1])` is faster than `amin(a, axis=0)`.

### Examples

```
>>> a = np.arange(4).reshape((2,2))
>>> a
array([[0, 1],
       [2, 3]])
>>> np.amin(a)           # Minimum of the flattened array
0
>>> np.amin(a, axis=0)   # Minima along the first axis
array([0, 1])
>>> np.amin(a, axis=1)   # Minima along the second axis
array([0, 2])
>>> np.amin(a, where=[False, True], initial=10, axis=0)
array([10,  1])
```

```
>>> b = np.arange(5, dtype=float)
>>> b[2] = np.NaN
>>> np.amin(b)
nan
>>> np.amin(b, where=~np.isnan(b), initial=10)
0.0
>>> np.nanmin(b)
0.0
```

```
>>> np.min([[-50], [10]], axis=-1, initial=0)
array([-50,   0])
```

Notice that the initial value is used as one of the elements for which the minimum is determined, unlike for the default argument Python's max function, which is only used for empty iterables.

Notice that this isn't the same as Python's `default` argument.

```
>>> np.min([6], initial=5)
5
>>> min([6], default=5)
6
```

`numpy.`**`amax`**(*a*, *axis=None*, *out=None*, *keepdims=<no value>*, *initial=<no value>*, *where=<no value>*)

Return the maximum of an array or maximum along an axis.

**Parameters**

**a**

[array_like] Input data.

**axis**

[None or int or tuple of ints, optional] Axis or axes along which to operate. By default, flattened input is used.

New in version 1.7.0.

If this is a tuple of ints, the maximum is selected over multiple axes, instead of a single axis or all the axes as before.

**out**

[ndarray, optional] Alternative output array in which to place the result. Must be of the same shape and buffer length as the expected output. See *ufuncs-output-type* for more details.

**keepdims**

[bool, optional] If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the input array.

If the default value is passed, then *keepdims* will not be passed through to the `amax` method of sub-classes of `ndarray`, however any non-default value will be. If the sub-class' method does not implement *keepdims* any exceptions will be raised.

**initial**

[scalar, optional] The minimum value of an output element. Must be present to allow computation on empty slice. See `reduce` for details.

New in version 1.15.0.

**where**

[array_like of bool, optional] Elements to compare for the maximum. See `reduce` for details.

New in version 1.17.0.

**Returns**

**amax**

[ndarray or scalar] Maximum of *a*. If *axis* is None, the result is a scalar value. If *axis* is given, the result is an array of dimension `a.ndim - 1`.

**See also:**

**`amin`**

The minimum value of an array along a given axis, propagating any NaNs.

**`nanmax`**

The maximum value of an array along a given axis, ignoring any NaNs.

**`maximum`**

Element-wise maximum of two arrays, propagating any NaNs.

**`fmax`**

Element-wise maximum of two arrays, ignoring any NaNs.

**`argmax`**

Return the indices of the maximum values.

`nanmin`, `minimum`, `fmin`

### Notes

NaN values are propagated, that is if at least one item is NaN, the corresponding max value will be NaN as well. To ignore NaN values (MATLAB behavior), please use nanmax.

Don't use *amax* for element-wise comparison of 2 arrays; when `a.shape[0]` is 2, `maximum(a[0], a[1])` is faster than `amax(a, axis=0)`.

### Examples

```
>>> a = np.arange(4).reshape((2,2))
>>> a
array([[0, 1],
       [2, 3]])
>>> np.amax(a)           # Maximum of the flattened array
3
>>> np.amax(a, axis=0)   # Maxima along the first axis
array([2, 3])
>>> np.amax(a, axis=1)   # Maxima along the second axis
array([1, 3])
>>> np.amax(a, where=[False, True], initial=-1, axis=0)
array([-1,  3])
>>> b = np.arange(5, dtype=float)
>>> b[2] = np.NaN
>>> np.amax(b)
nan
>>> np.amax(b, where=~np.isnan(b), initial=-1)
4.0
>>> np.nanmax(b)
4.0
```

You can use an initial value to compute the maximum of an empty slice, or to initialize it to a different value:

```
>>> np.max([[-50], [10]], axis=-1, initial=0)
array([ 0, 10])
```

Notice that the initial value is used as one of the elements for which the maximum is determined, unlike for the default argument Python's max function, which is only used for empty iterables.

```
>>> np.max([5], initial=6)
6
>>> max([5], default=6)
5
```

`numpy.`**`nanmin`**(*a*, *axis=None*, *out=None*, *keepdims=<no value>*)

Return minimum of an array or minimum along an axis, ignoring any NaNs. When all-NaN slices are encountered a `RuntimeWarning` is raised and Nan is returned for that slice.

**Parameters**

**a**

[array_like] Array containing numbers whose minimum is desired. If *a* is not an array, a conversion is attempted.

**axis**

[{int, tuple of int, None}, optional] Axis or axes along which the minimum is computed. The default is to compute the minimum of the flattened array.

**out**

[ndarray, optional] Alternate output array in which to place the result. The default is `None`; if provided, it must have the same shape as the expected output, but the type will be cast if necessary. See *ufuncs-output-type* for more details.

New in version 1.8.0.

**keepdims**

[bool, optional] If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the original *a*.

If the value is anything but the default, then *keepdims* will be passed through to the `min` method of sub-classes of `ndarray`. If the sub-classes methods does not implement *keepdims* any exceptions will be raised.

New in version 1.8.0.

**Returns**

**nanmin**

[ndarray] An array with the same shape as *a*, with the specified axis removed. If *a* is a 0-d array, or if axis is None, an ndarray scalar is returned. The same dtype as *a* is returned.

**See also:**

**`nanmax`**

The maximum value of an array along a given axis, ignoring any NaNs.

**`amin`**

The minimum value of an array along a given axis, propagating any NaNs.

**`fmin`**

Element-wise minimum of two arrays, ignoring any NaNs.

**`minimum`**

Element-wise minimum of two arrays, propagating any NaNs.

**`isnan`**

Shows which elements are Not a Number (NaN).

**`isfinite`**

Shows which elements are neither NaN nor infinity.

`amax`, `fmax`, `maximum`

## Notes

NumPy uses the IEEE Standard for Binary Floating-Point for Arithmetic (IEEE 754). This means that Not a Number is not equivalent to infinity. Positive infinity is treated as a very large number and negative infinity is treated as a very small (i.e. negative) number.

If the input has a integer type the function is equivalent to np.min.

## Examples

```
>>> a = np.array([[1, 2], [3, np.nan]])
>>> np.nanmin(a)
1.0
>>> np.nanmin(a, axis=0)
array([1.,  2.])
>>> np.nanmin(a, axis=1)
array([1.,  3.])
```

When positive infinity and negative infinity are present:

```
>>> np.nanmin([1, 2, np.nan, np.inf])
1.0
>>> np.nanmin([1, 2, np.nan, np.NINF])
-inf
```

`numpy.`**`nanmax`**(*a*, *axis=None*, *out=None*, *keepdims=<no value>*)

Return the maximum of an array or maximum along an axis, ignoring any NaNs. When all-NaN slices are encountered a `RuntimeWarning` is raised and NaN is returned for that slice.

**Parameters**

**a**

[array_like] Array containing numbers whose maximum is desired. If *a* is not an array, a conversion is attempted.

**axis**

[{int, tuple of int, None}, optional] Axis or axes along which the maximum is computed. The default is to compute the maximum of the flattened array.

**out**

[ndarray, optional] Alternate output array in which to place the result. The default is `None`; if provided, it must have the same shape as the expected output, but the type will be cast if necessary. See *ufuncs-output-type* for more details.

New in version 1.8.0.

**keepdims**

[bool, optional] If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the original *a*.

If the value is anything but the default, then *keepdims* will be passed through to the `max` method of sub-classes of `ndarray`. If the sub-classes methods does not implement *keepdims* any exceptions will be raised.

New in version 1.8.0.

**Returns**

**nanmax**

[ndarray] An array with the same shape as *a*, with the specified axis removed. If *a* is a 0-d array, or if axis is None, an ndarray scalar is returned. The same dtype as *a* is returned.

**See also:**

`nanmin`

The minimum value of an array along a given axis, ignoring any NaNs.

`amax`

The maximum value of an array along a given axis, propagating any NaNs.

`fmax`

Element-wise maximum of two arrays, ignoring any NaNs.

`maximum`

Element-wise maximum of two arrays, propagating any NaNs.

`isnan`

Shows which elements are Not a Number (NaN).

`isfinite`

Shows which elements are neither NaN nor infinity.

`amin`, `fmin`, `minimum`

## Notes

NumPy uses the IEEE Standard for Binary Floating-Point for Arithmetic (IEEE 754). This means that Not a Number is not equivalent to infinity. Positive infinity is treated as a very large number and negative infinity is treated as a very small (i.e. negative) number.

If the input has a integer type the function is equivalent to np.max.

## Examples

```
>>> a = np.array([[1, 2], [3, np.nan]])
>>> np.nanmax(a)
3.0
>>> np.nanmax(a, axis=0)
array([3.,  2.])
>>> np.nanmax(a, axis=1)
array([2.,  3.])
```

When positive infinity and negative infinity are present:

```
>>> np.nanmax([1, 2, np.nan, np.NINF])
2.0
>>> np.nanmax([1, 2, np.nan, np.inf])
inf
```

`numpy.`**`ptp`**`(`*a*, *axis=None*, *out=None*, *keepdims=<no value>*`)`

Range of values (maximum - minimum) along an axis.

The name of the function comes from the acronym for 'peak to peak'.

> **Warning:** `ptp` preserves the data type of the array. This means the return value for an input of signed integers with n bits (e.g. *np.int8*, *np.int16*, etc) is also a signed integer with n bits. In that case, peak-to-peak values greater than `2**(n-1)-1` will be returned as negative values. An example with a work-around is shown below.

**Parameters**

**a**

[array_like] Input values.

**axis**

[None or int or tuple of ints, optional] Axis along which to find the peaks. By default, flatten the array. *axis* may be negative, in which case it counts from the last to the first axis.

New in version 1.15.0.

If this is a tuple of ints, a reduction is performed on multiple axes, instead of a single axis or all the axes as before.

**out**

[array_like] Alternative output array in which to place the result. It must have the same shape and buffer length as the expected output, but the type of the output values will be cast if necessary.

**keepdims**

[bool, optional] If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the input array.

If the default value is passed, then *keepdims* will not be passed through to the `ptp` method of sub-classes of `ndarray`, however any non-default value will be. If the sub-class' method does not implement *keepdims* any exceptions will be raised.

**Returns**

**ptp**

[ndarray] A new array holding the result, unless *out* was specified, in which case a reference to *out* is returned.

### Examples

```
>>> x = np.array([[4, 9, 2, 10],
...               [6, 9, 7, 12]])
```

```
>>> np.ptp(x, axis=1)
array([8, 6])
```

```
>>> np.ptp(x, axis=0)
array([2, 0, 5, 2])
```

```
>>> np.ptp(x)
10
```

This example shows that a negative value can be returned when the input is an array of signed integers.

```
>>> y = np.array([[1, 127],
...               [0, 127],
...               [-1, 127],
...               [-2, 127]], dtype=np.int8)
>>> np.ptp(y, axis=1)
array([ 126,  127, -128, -127], dtype=int8)
```

A work-around is to use the *view()* method to view the result as unsigned integers with the same bit width:

```
>>> np.ptp(y, axis=1).view(np.uint8)
array([126, 127, 128, 129], dtype=uint8)
```

numpy.**percentile**(*a*, *q*, *axis=None*, *out=None*, *overwrite_input=False*, *interpolation='linear'*, *keepdims=False*)

Compute the q-th percentile of the data along the specified axis.

Returns the q-th percentile(s) of the array elements.

**Parameters**

**a**

[array_like] Input array or object that can be converted to an array.

**q**

[array_like of float] Percentile or sequence of percentiles to compute, which must be between 0 and 100 inclusive.

**axis**

[{int, tuple of int, None}, optional] Axis or axes along which the percentiles are computed. The default is to compute the percentile(s) along a flattened version of the array.

Changed in version 1.9.0: A tuple of axes is supported

**out**

[ndarray, optional] Alternative output array in which to place the result. It must have the same shape and buffer length as the expected output, but the type (of the output) will be cast if necessary.

**overwrite_input**

[bool, optional] If True, then allow the input array *a* to be modified by intermediate calculations, to save memory. In this case, the contents of the input *a* after this function completes is undefined.

**interpolation**

[{'linear', 'lower', 'higher', 'midpoint', 'nearest'}] This optional parameter specifies the interpolation method to use when the desired percentile lies between two data points `i < j`:

- 'linear': `i + (j - i) * fraction`, where `fraction` is the fractional part of the index surrounded by `i` and `j`.
- 'lower': `i`.
- 'higher': `j`.
- 'nearest': `i` or `j`, whichever is nearest.
- 'midpoint': `(i + j) / 2`.

New in version 1.9.0.

**keepdims**

[bool, optional] If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the original array *a*.

New in version 1.9.0.

**Returns**

**percentile**

[scalar or ndarray] If *q* is a single percentile and *axis=None*, then the result is a scalar. If multiple percentiles are given, first axis of the result corresponds to the percentiles. The other axes are the axes that remain after the reduction of *a*. If the input contains integers or floats smaller than `float64`, the output data-type is `float64`. Otherwise, the output data-type is the same as that of the input. If *out* is specified, that array is returned instead.

**See also:**

*`mean`*

***`median`***

equivalent to `percentile(..., 50)`

*`nanpercentile`*

***`quantile`***

equivalent to percentile, except with q in the range [0, 1].

## Notes

Given a vector `V` of length `N`, the q-th percentile of `V` is the value `q/100` of the way from the minimum to the maximum in a sorted copy of `V`. The values and distances of the two nearest neighbors as well as the *interpolation* parameter will determine the percentile if the normalized ranking does not match the location of `q` exactly. This function is the same as the median if `q=50`, the same as the minimum if `q=0` and the same as the maximum if `q=100`.

## Examples

```
>>> a = np.array([[10, 7, 4], [3, 2, 1]])
>>> a
array([[10,  7,  4],
       [ 3,  2,  1]])
>>> np.percentile(a, 50)
3.5
>>> np.percentile(a, 50, axis=0)
array([6.5, 4.5, 2.5])
>>> np.percentile(a, 50, axis=1)
array([7.,  2.])
>>> np.percentile(a, 50, axis=1, keepdims=True)
array([[7.],
       [2.]])
```

```
>>> m = np.percentile(a, 50, axis=0)
>>> out = np.zeros_like(m)
>>> np.percentile(a, 50, axis=0, out=out)
array([6.5, 4.5, 2.5])
>>> m
array([6.5, 4.5, 2.5])
```

```
>>> b = a.copy()
>>> np.percentile(b, 50, axis=1, overwrite_input=True)
array([7.,  2.])
>>> assert not np.all(a == b)
```

The different types of interpolation can be visualized graphically:

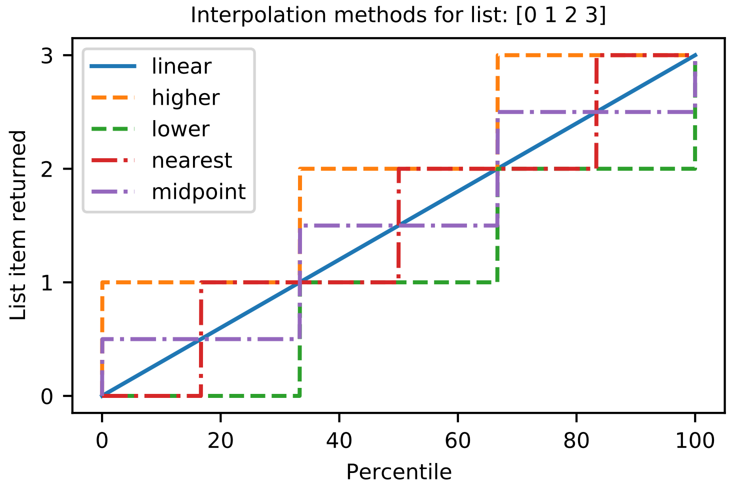
```
import matplotlib.pyplot as plt

a = np.arange(4)
p = np.linspace(0, 100, 6001)
ax = plt.gca()
lines = [
    ('linear', None),
    ('higher', '--'),
    ('lower', '--'),
    ('nearest', '-.'),
    ('midpoint', '-.'),
]
for interpolation, style in lines:
    ax.plot(
        p, np.percentile(a, p, interpolation=interpolation),
        label=interpolation, linestyle=style)
ax.set(
```

(continues on next page)

(continued from previous page)

```
    title='Interpolation methods for list: ' + str(a),
    xlabel='Percentile',
    ylabel='List item returned',
    yticks=a)
ax.legend()
plt.show()
```



`numpy.`**`nanpercentile`**(*a*, *q*, *axis=None*, *out=None*, *overwrite_input=False*, *interpolation='linear'*, *keepdims=<no value>*)

Compute the qth percentile of the data along the specified axis, while ignoring nan values.

Returns the qth percentile(s) of the array elements.

New in version 1.9.0.

**Parameters**

**a**

[array_like] Input array or object that can be converted to an array, containing nan values to be ignored.

**q**

[array_like of float] Percentile or sequence of percentiles to compute, which must be between 0 and 100 inclusive.

**axis**

[{int, tuple of int, None}, optional] Axis or axes along which the percentiles are computed. The default is to compute the percentile(s) along a flattened version of the array.

**out**

[ndarray, optional] Alternative output array in which to place the result. It must have the same shape and buffer length as the expected output, but the type (of the output) will be cast if necessary.

**overwrite_input**

[bool, optional] If True, then allow the input array *a* to be modified by intermediate calculations, to save memory. In this case, the contents of the input *a* after this function completes is undefined.

**interpolation**

[{'linear', 'lower', 'higher', 'midpoint', 'nearest'}] This optional parameter specifies the interpolation method to use when the desired percentile lies between two data points `i < j`:

- 'linear': `i + (j - i) * fraction`, where `fraction` is the fractional part of the index surrounded by `i` and `j`.
- 'lower': `i`.
- 'higher': `j`.
- 'nearest': `i` or `j`, whichever is nearest.
- 'midpoint': `(i + j) / 2`.

**keepdims**

[bool, optional] If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the original array *a*.

If this is anything but the default value it will be passed through (in the special case of an empty array) to the `mean` function of the underlying array. If the array is a sub-class and `mean` does not have the kwarg *keepdims* this will raise a RuntimeError.

**Returns**

**percentile**

[scalar or ndarray] If *q* is a single percentile and *axis=None*, then the result is a scalar. If multiple percentiles are given, first axis of the result corresponds to the percentiles. The other axes are the axes that remain after the reduction of *a*. If the input contains integers or floats smaller than `float64`, the output data-type is `float64`. Otherwise, the output data-type is the same as that of the input. If *out* is specified, that array is returned instead.

**See also:**

`nanmean`

**`nanmedian`**

equivalent to `nanpercentile(..., 50)`

`percentile`, `median`, `mean`

**`nanquantile`**

equivalent to nanpercentile, but with q in the range [0, 1].

### Notes

Given a vector `V` of length `N`, the `q`-th percentile of `V` is the value `q/100` of the way from the minimum to the maximum in a sorted copy of `V`. The values and distances of the two nearest neighbors as well as the *interpolation* parameter will determine the percentile if the normalized ranking does not match the location of `q` exactly. This function is the same as the median if `q=50`, the same as the minimum if `q=0` and the same as the maximum if `q=100`.

### Examples

```
>>> a = np.array([[10., 7., 4.], [3., 2., 1.]])
>>> a[0][1] = np.nan
>>> a
array([[10.,  nan,   4.],
      [ 3.,   2.,   1.]])
>>> np.percentile(a, 50)
nan
>>> np.nanpercentile(a, 50)
3.0
>>> np.nanpercentile(a, 50, axis=0)
array([6.5, 2. , 2.5])
>>> np.nanpercentile(a, 50, axis=1, keepdims=True)
array([[7.],
       [2.]])
>>> m = np.nanpercentile(a, 50, axis=0)
>>> out = np.zeros_like(m)
>>> np.nanpercentile(a, 50, axis=0, out=out)
array([6.5, 2. , 2.5])
>>> m
array([6.5,  2. ,  2.5])
```

```
>>> b = a.copy()
>>> np.nanpercentile(b, 50, axis=1, overwrite_input=True)
array([7., 2.])
>>> assert not np.all(a==b)
```

`numpy.`**`quantile`**(*a*, *q*, *axis=None*, *out=None*, *overwrite_input=False*, *interpolation='linear'*, *keepdims=False*)

Compute the q-th quantile of the data along the specified axis.

New in version 1.15.0.

**Parameters**

**a**

[array_like] Input array or object that can be converted to an array.

**q**

[array_like of float] Quantile or sequence of quantiles to compute, which must be between 0 and 1 inclusive.

**axis**

[{int, tuple of int, None}, optional] Axis or axes along which the quantiles are computed. The default is to compute the quantile(s) along a flattened version of the array.

**out**

[ndarray, optional] Alternative output array in which to place the result. It must have the same shape and buffer length as the expected output, but the type (of the output) will be cast if necessary.

**overwrite_input**

[bool, optional] If True, then allow the input array *a* to be modified by intermediate calculations, to save memory. In this case, the contents of the input *a* after this function completes is undefined.

**interpolation**

[{'linear', 'lower', 'higher', 'midpoint', 'nearest'}] This optional parameter specifies the interpolation method to use when the desired quantile lies between two data points `i < j`:

- linear: `i + (j - i) * fraction`, where `fraction` is the fractional part of the index surrounded by `i` and `j`.
- lower: `i`.
- higher: `j`.
- nearest: `i` or `j`, whichever is nearest.
- midpoint: `(i + j) / 2`.

**keepdims**

[bool, optional] If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the original array *a*.

**Returns**

**quantile**

[scalar or ndarray] If *q* is a single quantile and *axis=None*, then the result is a scalar. If multiple quantiles are given, first axis of the result corresponds to the quantiles. The other axes are the axes that remain after the reduction of *a*. If the input contains integers or floats smaller than `float64`, the output data-type is `float64`. Otherwise, the output data-type is the same as that of the input. If *out* is specified, that array is returned instead.

**See also:**

`mean`

**`percentile`**

equivalent to quantile, but with q in the range [0, 100].

**`median`**

equivalent to `quantile(..., 0.5)`

`nanquantile`

## Notes

Given a vector `V` of length `N`, the q-th quantile of `V` is the value `q` of the way from the minimum to the maximum in a sorted copy of `V`. The values and distances of the two nearest neighbors as well as the *interpolation* parameter will determine the quantile if the normalized ranking does not match the location of `q` exactly. This function is the same as the median if `q=0.5`, the same as the minimum if `q=0.0` and the same as the maximum if `q=1.0`.

## Examples

```
>>> a = np.array([[10, 7, 4], [3, 2, 1]])
>>> a
array([[10,  7,  4],
       [ 3,  2,  1]])
>>> np.quantile(a, 0.5)
3.5
>>> np.quantile(a, 0.5, axis=0)
array([6.5, 4.5, 2.5])
>>> np.quantile(a, 0.5, axis=1)
array([7.,  2.])
>>> np.quantile(a, 0.5, axis=1, keepdims=True)
array([[7.],
       [2.]])
>>> m = np.quantile(a, 0.5, axis=0)
>>> out = np.zeros_like(m)
>>> np.quantile(a, 0.5, axis=0, out=out)
array([6.5, 4.5, 2.5])
>>> m
array([6.5, 4.5, 2.5])
>>> b = a.copy()
>>> np.quantile(b, 0.5, axis=1, overwrite_input=True)
array([7.,  2.])
>>> assert not np.all(a == b)
```

`numpy.`**`nanquantile`**(*a*, *q*, *axis=None*, *out=None*, *overwrite_input=False*, *interpolation='linear'*, *keepdims=<no value>*)

Compute the qth quantile of the data along the specified axis, while ignoring nan values. Returns the qth quantile(s) of the array elements.

New in version 1.15.0.

**Parameters**

**a**

[array_like] Input array or object that can be converted to an array, containing nan values to be ignored

**q**

[array_like of float] Quantile or sequence of quantiles to compute, which must be between 0 and 1 inclusive.

**axis**

[{int, tuple of int, None}, optional] Axis or axes along which the quantiles are computed. The default is to compute the quantile(s) along a flattened version of the array.

**out**

[ndarray, optional] Alternative output array in which to place the result. It must have the same shape and buffer length as the expected output, but the type (of the output) will be cast if necessary.

**overwrite_input**

[bool, optional] If True, then allow the input array *a* to be modified by intermediate calculations, to save memory. In this case, the contents of the input *a* after this function completes is undefined.

**interpolation**

[{'linear', 'lower', 'higher', 'midpoint', 'nearest'}] This optional parameter specifies the interpolation method to use when the desired quantile lies between two data points `i < j`:

- linear: `i + (j - i) * fraction`, where `fraction` is the fractional part of the index surrounded by `i` and `j`.
- lower: `i`.
- higher: `j`.
- nearest: `i` or `j`, whichever is nearest.
- midpoint: `(i + j) / 2`.

**keepdims**

[bool, optional] If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the original array *a*.

If this is anything but the default value it will be passed through (in the special case of an empty array) to the `mean` function of the underlying array. If the array is a sub-class and `mean` does not have the kwarg *keepdims* this will raise a RuntimeError.

**Returns**

**quantile**

[scalar or ndarray] If *q* is a single percentile and *axis=None*, then the result is a scalar. If multiple quantiles are given, first axis of the result corresponds to the quantiles. The other axes are the axes that remain after the reduction of *a*. If the input contains integers or floats smaller than `float64`, the output data-type is `float64`. Otherwise, the output data-type is the same as that of the input. If *out* is specified, that array is returned instead.

**See also:**

`quantile`, `nanmean`, `nanmedian`

**`nanmedian`**

equivalent to `nanquantile(..., 0.5)`

**`nanpercentile`**

same as nanquantile, but with q in the range [0, 100].

### Examples

```
>>> a = np.array([[10., 7., 4.], [3., 2., 1.]])
>>> a[0][1] = np.nan
>>> a
array([[10.,  nan,   4.],
      [ 3.,   2.,   1.]])
>>> np.quantile(a, 0.5)
nan
>>> np.nanquantile(a, 0.5)
3.0
>>> np.nanquantile(a, 0.5, axis=0)
array([6.5, 2. , 2.5])
>>> np.nanquantile(a, 0.5, axis=1, keepdims=True)
array([[7.],
       [2.]])
>>> m = np.nanquantile(a, 0.5, axis=0)
>>> out = np.zeros_like(m)
>>> np.nanquantile(a, 0.5, axis=0, out=out)
array([6.5, 2. , 2.5])
>>> m
array([6.5,  2. ,  2.5])
>>> b = a.copy()
>>> np.nanquantile(b, 0.5, axis=1, overwrite_input=True)
array([7., 2.])
>>> assert not np.all(a==b)
```

## 4.27.2 Averages and variances

| | |
|---|---|
| `median`(a[, axis, out, overwrite_input, keepdims]) | Compute the median along the specified axis. |
| `average`(a[, axis, weights, returned]) | Compute the weighted average along the specified axis. |
| `mean`(a[, axis, dtype, out, keepdims]) | Compute the arithmetic mean along the specified axis. |
| `std`(a[, axis, dtype, out, ddof, keepdims]) | Compute the standard deviation along the specified axis. |
| `var`(a[, axis, dtype, out, ddof, keepdims]) | Compute the variance along the specified axis. |
| `nanmedian`(a[, axis, out, overwrite_input, …]) | Compute the median along the specified axis, while ignoring NaNs. |
| `nanmean`(a[, axis, dtype, out, keepdims]) | Compute the arithmetic mean along the specified axis, ignoring NaNs. |
| `nanstd`(a[, axis, dtype, out, ddof, keepdims]) | Compute the standard deviation along the specified axis, while ignoring NaNs. |
| `nanvar`(a[, axis, dtype, out, ddof, keepdims]) | Compute the variance along the specified axis, while ignoring NaNs. |

`numpy.`**`median`**(*a*, *axis=None*, *out=None*, *overwrite_input=False*, *keepdims=False*)

Compute the median along the specified axis.

Returns the median of the array elements.

**Parameters**

**a**

[array_like] Input array or object that can be converted to an array.

**axis**

[{int, sequence of int, None}, optional] Axis or axes along which the medians are computed. The default is to compute the median along a flattened version of the array. A sequence of axes is supported since version 1.9.0.

**out**

[ndarray, optional] Alternative output array in which to place the result. It must have the same shape and buffer length as the expected output, but the type (of the output) will be cast if necessary.

**overwrite_input**

[bool, optional] If True, then allow use of memory of input array *a* for calculations. The input array will be modified by the call to *`median`*. This will save memory when you do not need to preserve the contents of the input array. Treat the input as undefined, but it will probably be fully or partially sorted. Default is False. If *overwrite_input* is `True` and *a* is not already an *`ndarray`*, an error will be raised.

**keepdims**

[bool, optional] If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the original *arr*.

New in version 1.9.0.

**Returns**

**median**

[ndarray] A new array holding the result. If the input contains integers or floats smaller than `float64`, then the output data-type is `np.float64`. Otherwise, the data-type of the output is the same as that of the input. If *out* is specified, that array is returned instead.

**See also:**

*`mean`*, *`percentile`*

## Notes

Given a vector `V` of length `N`, the median of `V` is the middle value of a sorted copy of `V`, `V_sorted` - i e., `V_sorted[(N-1)/2]`, when `N` is odd, and the average of the two middle values of `V_sorted` when `N` is even.

## Examples

```
>>> a = np.array([[10, 7, 4], [3, 2, 1]])
>>> a
array([[10,  7,  4],
       [ 3,  2,  1]])
>>> np.median(a)
3.5
>>> np.median(a, axis=0)
array([6.5, 4.5, 2.5])
>>> np.median(a, axis=1)
array([7.,  2.])
>>> m = np.median(a, axis=0)
```

(continues on next page)

(continued from previous page)

```
>>> out = np.zeros_like(m)
>>> np.median(a, axis=0, out=m)
array([6.5,  4.5,  2.5])
>>> m
array([6.5,  4.5,  2.5])
>>> b = a.copy()
>>> np.median(b, axis=1, overwrite_input=True)
array([7.,  2.])
>>> assert not np.all(a==b)
>>> b = a.copy()
>>> np.median(b, axis=None, overwrite_input=True)
3.5
>>> assert not np.all(a==b)
```

`numpy.`**`average`**(*a*, *axis=None*, *weights=None*, *returned=False*)

Compute the weighted average along the specified axis.

**Parameters**

**a**

[array_like] Array containing data to be averaged. If *a* is not an array, a conversion is attempted.

**axis**

[None or int or tuple of ints, optional] Axis or axes along which to average *a*. The default, axis=None, will average over all of the elements of the input array. If axis is negative it counts from the last to the first axis.

New in version 1.7.0.

If axis is a tuple of ints, averaging is performed on all of the axes specified in the tuple instead of a single axis or all the axes as before.

**weights**

[array_like, optional] An array of weights associated with the values in *a*. Each value in *a* contributes to the average according to its associated weight. The weights array can either be 1-D (in which case its length must be the size of *a* along the given axis) or of the same shape as *a*. If *weights=None*, then all data in *a* are assumed to have a weight equal to one. The 1-D calculation is:

```
avg = sum(a * weights) / sum(weights)
```

The only constraint on *weights* is that *sum(weights)* must not be 0.

**returned**

[bool, optional] Default is *False*. If *True*, the tuple (`average`, *sum_of_weights*) is returned, otherwise only the average is returned. If *weights=None*, *sum_of_weights* is equivalent to the number of elements over which the average is taken.

**Returns**

**retval, [sum_of_weights]**

[array_type or double] Return the average along the specified axis. When *returned* is *True*, return a tuple with the average as the first element and the sum of the weights as the second element. *sum_of_weights* is of the same type as *retval*. The result dtype follows a genereal pattern. If *weights* is None, the result dtype will be that of *a* , or `float64` if *a* is integral. Otherwise, if *weights* is not None and *a* is non- integral, the result type will be the type of lowest precision capable of representing values of both *a* and *weights*. If *a* happens to be integral, the previous rules still applies but the result dtype will at least be `float64`.

**Raises**

**ZeroDivisionError**

When all weights along axis are zero. See `numpy.ma.average` for a version robust to this type of error.

**TypeError**

When the length of 1D *weights* is not the same as the shape of *a* along axis.

**See also:**

`mean`

**`ma.average`**

average for masked arrays – useful if your data contains "missing" values

**`numpy.result_type`**

Returns the type that results from applying the numpy type promotion rules to the arguments.

## Examples

```
>>> data = np.arange(1, 5)
>>> data
array([1, 2, 3, 4])
>>> np.average(data)
2.5
>>> np.average(np.arange(1, 11), weights=np.arange(10, 0, -1))
4.0
```

```
>>> data = np.arange(6).reshape((3,2))
>>> data
array([[0, 1],
       [2, 3],
       [4, 5]])
>>> np.average(data, axis=1, weights=[1./4, 3./4])
array([0.75, 2.75, 4.75])
>>> np.average(data, weights=[1./4, 3./4])
Traceback (most recent call last):
    ...
TypeError: Axis must be specified when shapes of a and weights differ.
```

```
>>> a = np.ones(5, dtype=np.float128)
>>> w = np.ones(5, dtype=np.complex64)
>>> avg = np.average(a, weights=w)
>>> print(avg.dtype)
complex256
```

`numpy.mean`(*a*, *axis=None*, *dtype=None*, *out=None*, *keepdims=<no value>*)

Compute the arithmetic mean along the specified axis.

Returns the average of the array elements. The average is taken over the flattened array by default, otherwise over the specified axis. `float64` intermediate and return values are used for integer inputs.

**Parameters**

**a**

[array_like] Array containing numbers whose mean is desired. If *a* is not an array, a conversion is attempted.

**axis**

[None or int or tuple of ints, optional] Axis or axes along which the means are computed. The default is to compute the mean of the flattened array.

New in version 1.7.0.

If this is a tuple of ints, a mean is performed over multiple axes, instead of a single axis or all the axes as before.

**dtype**

[data-type, optional] Type to use in computing the mean. For integer inputs, the default is `float64`; for floating point inputs, it is the same as the input dtype.

**out**

[ndarray, optional] Alternate output array in which to place the result. The default is `None`; if provided, it must have the same shape as the expected output, but the type will be cast if necessary. See *ufuncs-output-type* for more details.

**keepdims**

[bool, optional] If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the input array.

If the default value is passed, then *keepdims* will not be passed through to the `mean` method of sub-classes of `ndarray`, however any non-default value will be. If the sub-class' method does not implement *keepdims* any exceptions will be raised.

**Returns**

**m**

[ndarray, see dtype parameter above] If *out=None*, returns a new array containing the mean values, otherwise a reference to the output array is returned.

**See also:**

**`average`**

Weighted average

`std`, `var`, `nanmean`, `nanstd`, `nanvar`

### Notes

The arithmetic mean is the sum of the elements along the axis divided by the number of elements.

Note that for floating-point input, the mean is computed using the same precision the input has. Depending on the input data, this can cause the results to be inaccurate, especially for `float32` (see example below). Specifying a higher-precision accumulator using the *`dtype`* keyword can alleviate this issue.

By default, `float16` results are computed using `float32` intermediates for extra precision.

### Examples

```
>>> a = np.array([[1, 2], [3, 4]])
>>> np.mean(a)
2.5
>>> np.mean(a, axis=0)
array([2., 3.])
>>> np.mean(a, axis=1)
array([1.5, 3.5])
```

In single precision, *`mean`* can be inaccurate:

```
>>> a = np.zeros((2, 512*512), dtype=np.float32)
>>> a[0, :] = 1.0
>>> a[1, :] = 0.1
>>> np.mean(a)
0.54999924
```

Computing the mean in float64 is more accurate:

```
>>> np.mean(a, dtype=np.float64)
0.55000000074505806 # may vary
```

`numpy.`**`std`**`(`*a*, *axis=None*, *dtype=None*, *out=None*, *ddof=0*, *keepdims=<no value>*`)`

Compute the standard deviation along the specified axis.

Returns the standard deviation, a measure of the spread of a distribution, of the array elements. The standard deviation is computed for the flattened array by default, otherwise over the specified axis.

**Parameters**

**a**

[array_like] Calculate the standard deviation of these values.

**axis**

[None or int or tuple of ints, optional] Axis or axes along which the standard deviation is computed. The default is to compute the standard deviation of the flattened array.

New in version 1.7.0.

If this is a tuple of ints, a standard deviation is performed over multiple axes, instead of a single axis or all the axes as before.

**dtype**

[dtype, optional] Type to use in computing the standard deviation. For arrays of integer type the default is float64, for arrays of float types it is the same as the array type.

**out**

[ndarray, optional] Alternative output array in which to place the result. It must have the same shape as the expected output but the type (of the calculated values) will be cast if necessary.

**ddof**

[int, optional] Means Delta Degrees of Freedom. The divisor used in calculations is `N - ddof`, where `N` represents the number of elements. By default *ddof* is zero.

**keepdims**

[bool, optional] If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the input array.

If the default value is passed, then *keepdims* will not be passed through to the `std` method of sub-classes of `ndarray`, however any non-default value will be. If the sub-class' method does not implement *keepdims* any exceptions will be raised.

**Returns**

**standard_deviation**

[ndarray, see dtype parameter above.] If *out* is None, return a new array containing the standard deviation, otherwise return a reference to the output array.

**See also:**

`var`, `mean`, `nanmean`, `nanstd`, `nanvar`, `ufuncs-output-type`

### Notes

The standard deviation is the square root of the average of the squared deviations from the mean, i.e., `std = sqrt(mean(abs(x - x.mean())**2))`.

The average squared deviation is normally calculated as `x.sum() / N`, where `N = len(x)`. If, however, *ddof* is specified, the divisor `N - ddof` is used instead. In standard statistical practice, `ddof=1` provides an unbiased estimator of the variance of the infinite population. `ddof=0` provides a maximum likelihood estimate of the variance for normally distributed variables. The standard deviation computed in this function is the square root of the estimated variance, so even with `ddof=1`, it will not be an unbiased estimate of the standard deviation per se.

Note that, for complex numbers, `std` takes the absolute value before squaring, so that the result is always real and nonnegative.

For floating-point input, the *std* is computed using the same precision the input has. Depending on the input data, this can cause the results to be inaccurate, especially for float32 (see example below). Specifying a higher-accuracy accumulator using the `dtype` keyword can alleviate this issue.

## Examples

```
>>> a = np.array([[1, 2], [3, 4]])
>>> np.std(a)
1.1180339887498949 # may vary
>>> np.std(a, axis=0)
array([1.,  1.])
>>> np.std(a, axis=1)
array([0.5,  0.5])
```

In single precision, std() can be inaccurate:

```
>>> a = np.zeros((2, 512*512), dtype=np.float32)
>>> a[0, :] = 1.0
>>> a[1, :] = 0.1
>>> np.std(a)
0.45000005
```

Computing the standard deviation in float64 is more accurate:

```
>>> np.std(a, dtype=np.float64)
0.44999999925494177 # may vary
```

`numpy.`**`var`**(*a*, *axis=None*, *dtype=None*, *out=None*, *ddof=0*, *keepdims=<no value>*)

Compute the variance along the specified axis.

Returns the variance of the array elements, a measure of the spread of a distribution. The variance is computed for the flattened array by default, otherwise over the specified axis.

**Parameters**

**a**

[array_like] Array containing numbers whose variance is desired. If *a* is not an array, a conversion is attempted.

**axis**

[None or int or tuple of ints, optional] Axis or axes along which the variance is computed. The default is to compute the variance of the flattened array.

New in version 1.7.0.

If this is a tuple of ints, a variance is performed over multiple axes, instead of a single axis or all the axes as before.

**dtype**

[data-type, optional] Type to use in computing the variance. For arrays of integer type the default is `float64`; for arrays of float types it is the same as the array type.

**out**

[ndarray, optional] Alternate output array in which to place the result. It must have the same shape as the expected output, but the type is cast if necessary.

**ddof**

[int, optional] “Delta Degrees of Freedom”: the divisor used in the calculation is `N - ddof`, where `N` represents the number of elements. By default *ddof* is zero.

**keepdims**

[bool, optional] If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the input array.

If the default value is passed, then *keepdims* will not be passed through to the `var` method of sub-classes of `ndarray`, however any non-default value will be. If the sub-class' method does not implement *keepdims* any exceptions will be raised.

**Returns**

**variance**

[ndarray, see dtype parameter above] If `out=None`, returns a new array containing the variance; otherwise, a reference to the output array is returned.

**See also:**

`std`, `mean`, `nanmean`, `nanstd`, `nanvar`, `ufuncs-output-type`

## Notes

The variance is the average of the squared deviations from the mean, i.e., `var = mean(abs(x - x.mean())**2)`.

The mean is normally calculated as `x.sum() / N`, where `N = len(x)`. If, however, *ddof* is specified, the divisor `N - ddof` is used instead. In standard statistical practice, `ddof=1` provides an unbiased estimator of the variance of a hypothetical infinite population. `ddof=0` provides a maximum likelihood estimate of the variance for normally distributed variables.

Note that for complex numbers, the absolute value is taken before squaring, so that the result is always real and nonnegative.

For floating-point input, the variance is computed using the same precision the input has. Depending on the input data, this can cause the results to be inaccurate, especially for `float32` (see example below). Specifying a higher-accuracy accumulator using the `dtype` keyword can alleviate this issue.

## Examples

```
>>> a = np.array([[1, 2], [3, 4]])
>>> np.var(a)
1.25
>>> np.var(a, axis=0)
array([1.,  1.])
>>> np.var(a, axis=1)
array([0.25,  0.25])
```

In single precision, var() can be inaccurate:

```
>>> a = np.zeros((2, 512*512), dtype=np.float32)
>>> a[0, :] = 1.0
>>> a[1, :] = 0.1
>>> np.var(a)
0.20250003
```

Computing the variance in float64 is more accurate:

```
>>> np.var(a, dtype=np.float64)
0.20249999932944759 # may vary
>>> ((1-0.55)**2 + (0.1-0.55)**2)/2
0.2025
```

`numpy.`**`nanmedian`**(*a*, *axis=None*, *out=None*, *overwrite_input=False*, *keepdims=<no value>*)

Compute the median along the specified axis, while ignoring NaNs.

Returns the median of the array elements.

New in version 1.9.0.

**Parameters**

**a**

[array_like] Input array or object that can be converted to an array.

**axis**

[{int, sequence of int, None}, optional] Axis or axes along which the medians are computed. The default is to compute the median along a flattened version of the array. A sequence of axes is supported since version 1.9.0.

**out**

[ndarray, optional] Alternative output array in which to place the result. It must have the same shape and buffer length as the expected output, but the type (of the output) will be cast if necessary.

**overwrite_input**

[bool, optional] If True, then allow use of memory of input array *a* for calculations. The input array will be modified by the call to `median`. This will save memory when you do not need to preserve the contents of the input array. Treat the input as undefined, but it will probably be fully or partially sorted. Default is False. If *overwrite_input* is `True` and *a* is not already an `ndarray`, an error will be raised.

**keepdims**

[bool, optional] If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the original *a*.

If this is anything but the default value it will be passed through (in the special case of an empty array) to the `mean` function of the underlying array. If the array is a sub-class and `mean` does not have the kwarg *keepdims* this will raise a RuntimeError.

**Returns**

**median**

[ndarray] A new array holding the result. If the input contains integers or floats smaller than `float64`, then the output data-type is `np.float64`. Otherwise, the data-type of the output is the same as that of the input. If *out* is specified, that array is returned instead.

**See also:**

`mean`, `median`, `percentile`

### Notes

Given a vector `V` of length `N`, the median of `V` is the middle value of a sorted copy of `V`, `V_sorted` - i.e., `V_sorted[(N-1)/2]`, when `N` is odd and the average of the two middle values of `V_sorted` when `N` is even.

### Examples

```
>>> a = np.array([[10.0, 7, 4], [3, 2, 1]])
>>> a[0, 1] = np.nan
>>> a
array([[10., nan,  4.],
       [ 3.,  2.,  1.]])
>>> np.median(a)
nan
>>> np.nanmedian(a)
3.0
>>> np.nanmedian(a, axis=0)
array([6.5, 2. , 2.5])
>>> np.median(a, axis=1)
array([nan,  2.])
>>> b = a.copy()
>>> np.nanmedian(b, axis=1, overwrite_input=True)
array([7.,  2.])
>>> assert not np.all(a==b)
>>> b = a.copy()
>>> np.nanmedian(b, axis=None, overwrite_input=True)
3.0
>>> assert not np.all(a==b)
```

`numpy.`**`nanmean`**(*a*, *axis=None*, *dtype=None*, *out=None*, *keepdims=<no value>*)

Compute the arithmetic mean along the specified axis, ignoring NaNs.

Returns the average of the array elements. The average is taken over the flattened array by default, otherwise over the specified axis. `float64` intermediate and return values are used for integer inputs.

For all-NaN slices, NaN is returned and a *RuntimeWarning* is raised.

New in version 1.8.0.

**Parameters**

**a**

[array_like] Array containing numbers whose mean is desired. If *a* is not an array, a conversion is attempted.

**axis**

[{int, tuple of int, None}, optional] Axis or axes along which the means are computed. The default is to compute the mean of the flattened array.

**dtype**

[data-type, optional] Type to use in computing the mean. For integer inputs, the default is `float64`; for inexact inputs, it is the same as the input dtype.

**out**

[ndarray, optional] Alternate output array in which to place the result. The default is `None`; if provided, it must have the same shape as the expected output, but the type will be cast if necessary. See *ufuncs-output-type* for more details.

**keepdims**

[bool, optional] If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the original *a*.

If the value is anything but the default, then *keepdims* will be passed through to the `mean` or `sum` methods of sub-classes of `ndarray`. If the sub-classes methods does not implement *keepdims* any exceptions will be raised.

**Returns**

**m**

[ndarray, see dtype parameter above] If *out=None*, returns a new array containing the mean values, otherwise a reference to the output array is returned. Nan is returned for slices that contain only NaNs.

**See also:**

**`average`**

Weighted average

**`mean`**

Arithmetic mean taken while not ignoring NaNs

`var`, `nanvar`

## Notes

The arithmetic mean is the sum of the non-NaN elements along the axis divided by the number of non-NaN elements.

Note that for floating-point input, the mean is computed using the same precision the input has. Depending on the input data, this can cause the results to be inaccurate, especially for `float32`. Specifying a higher-precision accumulator using the `dtype` keyword can alleviate this issue.

## Examples

```
>>> a = np.array([[1, np.nan], [3, 4]])
>>> np.nanmean(a)
2.6666666666666665
>>> np.nanmean(a, axis=0)
array([2.,  4.])
>>> np.nanmean(a, axis=1)
array([1.,  3.5]) # may vary
```

`numpy.`**`nanstd`**(*a*, *axis=None*, *dtype=None*, *out=None*, *ddof=0*, *keepdims=<no value>*)

Compute the standard deviation along the specified axis, while ignoring NaNs.

Returns the standard deviation, a measure of the spread of a distribution, of the non-NaN array elements. The standard deviation is computed for the flattened array by default, otherwise over the specified axis.

For all-NaN slices or slices with zero degrees of freedom, NaN is returned and a *RuntimeWarning* is raised.

New in version 1.8.0.

**Parameters**

**a**

[array_like] Calculate the standard deviation of the non-NaN values.

**axis**

[{int, tuple of int, None}, optional] Axis or axes along which the standard deviation is computed. The default is to compute the standard deviation of the flattened array.

**dtype**

[dtype, optional] Type to use in computing the standard deviation. For arrays of integer type the default is float64, for arrays of float types it is the same as the array type.

**out**

[ndarray, optional] Alternative output array in which to place the result. It must have the same shape as the expected output but the type (of the calculated values) will be cast if necessary.

**ddof**

[int, optional] Means Delta Degrees of Freedom. The divisor used in calculations is `N - ddof`, where `N` represents the number of non-NaN elements. By default *ddof* is zero.

**keepdims**

[bool, optional] If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the original *a*.

If this value is anything but the default it is passed through as-is to the relevant functions of the sub-classes. If these functions do not have a *keepdims* kwarg, a RuntimeError will be raised.

**Returns**

**standard_deviation**

[ndarray, see dtype parameter above.] If *out* is None, return a new array containing the standard deviation, otherwise return a reference to the output array. If ddof is >= the number of non-NaN elements in a slice or the slice contains only NaNs, then the result for that slice is NaN.

**See also:**

*`var`*, *`mean`*, *`std`*, *`nanvar`*, *`nanmean`*, `ufuncs-output-type`

### Notes

The standard deviation is the square root of the average of the squared deviations from the mean: `std = sqrt(mean(abs(x - x.mean())**2))`.

The average squared deviation is normally calculated as `x.sum() / N`, where `N = len(x)`. If, however, *ddof* is specified, the divisor `N - ddof` is used instead. In standard statistical practice, `ddof=1` provides an unbiased estimator of the variance of the infinite population. `ddof=0` provides a maximum likelihood estimate of the variance for normally distributed variables. The standard deviation computed in this function is the square root of the estimated variance, so even with `ddof=1`, it will not be an unbiased estimate of the standard deviation per se.

Note that, for complex numbers, `std` takes the absolute value before squaring, so that the result is always real and nonnegative.

For floating-point input, the *std* is computed using the same precision the input has. Depending on the input data, this can cause the results to be inaccurate, especially for float32 (see example below). Specifying a higher-accuracy accumulator using the `dtype` keyword can alleviate this issue.

### Examples

```
>>> a = np.array([[1, np.nan], [3, 4]])
>>> np.nanstd(a)
1.247219128924647
>>> np.nanstd(a, axis=0)
array([1., 0.])
>>> np.nanstd(a, axis=1)
array([0.,  0.5]) # may vary
```

`numpy.`**`nanvar`**(*a*, *axis=None*, *dtype=None*, *out=None*, *ddof=0*, *keepdims=<no value>*)

Compute the variance along the specified axis, while ignoring NaNs.

Returns the variance of the array elements, a measure of the spread of a distribution. The variance is computed for the flattened array by default, otherwise over the specified axis.

For all-NaN slices or slices with zero degrees of freedom, NaN is returned and a *RuntimeWarning* is raised.

New in version 1.8.0.

**Parameters**

**a**

[array_like] Array containing numbers whose variance is desired. If *a* is not an array, a conversion is attempted.

**axis**

[{int, tuple of int, None}, optional] Axis or axes along which the variance is computed. The default is to compute the variance of the flattened array.

**dtype**

[data-type, optional] Type to use in computing the variance. For arrays of integer type the default is `float64`; for arrays of float types it is the same as the array type.

**out**

[ndarray, optional] Alternate output array in which to place the result. It must have the same shape as the expected output, but the type is cast if necessary.

**ddof**

[int, optional] "Delta Degrees of Freedom": the divisor used in the calculation is `N - ddof`, where `N` represents the number of non-NaN elements. By default *ddof* is zero.

**keepdims**

[bool, optional] If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the original *a*.

**Returns**

**variance**

[ndarray, see dtype parameter above] If *out* is None, return a new array containing the variance, otherwise return a reference to the output array. If ddof is >= the number of non-NaN elements in a slice or the slice contains only NaNs, then the result for that slice is NaN.

**See also:**

**`std`**

Standard deviation

**`mean`**

Average

**`var`**

Variance while not ignoring NaNs

`nanstd`, `nanmean`, `ufuncs-output-type`

## Notes

The variance is the average of the squared deviations from the mean, i.e., `var = mean(abs(x - x.mean())**2)`.

The mean is normally calculated as `x.sum() / N`, where `N = len(x)`. If, however, *ddof* is specified, the divisor `N - ddof` is used instead. In standard statistical practice, `ddof=1` provides an unbiased estimator of the variance of a hypothetical infinite population. `ddof=0` provides a maximum likelihood estimate of the variance for normally distributed variables.

Note that for complex numbers, the absolute value is taken before squaring, so that the result is always real and nonnegative.

For floating-point input, the variance is computed using the same precision the input has. Depending on the input data, this can cause the results to be inaccurate, especially for `float32` (see example below). Specifying a higher-accuracy accumulator using the `dtype` keyword can alleviate this issue.

For this function to work on sub-classes of ndarray, they must define `sum` with the kwarg *keepdims*

### Examples

```
>>> a = np.array([[1, np.nan], [3, 4]])
>>> np.nanvar(a)
1.5555555555555554
>>> np.nanvar(a, axis=0)
array([1.,  0.])
>>> np.nanvar(a, axis=1)
array([0.,  0.25])  # may vary
```

## 4.27.3 Correlating

| | |
|---|---|
| `corrcoef`(x[, y, rowvar, bias, ddof]) | Return Pearson product-moment correlation coefficients. |
| `correlate`(a, v[, mode]) | Cross-correlation of two 1-dimensional sequences. |
| `cov`(m[, y, rowvar, bias, ddof, fweights, …]) | Estimate a covariance matrix, given data and weights. |

`numpy.`**`corrcoef`**(*x*, *y=None*, *rowvar=True*, *bias=<no value>*, *ddof=<no value>*)

Return Pearson product-moment correlation coefficients.

Please refer to the documentation for `cov` for more detail. The relationship between the correlation coefficient matrix, *R*, and the covariance matrix, *C*, is

$$R_{ij} = \frac{C_{ij}}{\sqrt{C_{ii} * C_{jj}}}$$

The values of *R* are between -1 and 1, inclusive.

**Parameters**

**x**

[array_like] A 1-D or 2-D array containing multiple variables and observations. Each row of *x* represents a variable, and each column a single observation of all those variables. Also see *rowvar* below.

**y**

[array_like, optional] An additional set of variables and observations. *y* has the same shape as *x*.

**rowvar**

[bool, optional] If *rowvar* is True (default), then each row represents a variable, with observations in the columns. Otherwise, the relationship is transposed: each column represents a variable, while the rows contain observations.

**bias**

[_NoValue, optional] Has no effect, do not use.

Deprecated since version 1.10.0.

**ddof**

[_NoValue, optional] Has no effect, do not use.

Deprecated since version 1.10.0.

**Returns**

**R**

[ndarray] The correlation coefficient matrix of the variables.

**See also:**

**`cov`**

Covariance matrix

### Notes

Due to floating point rounding the resulting array may not be Hermitian, the diagonal elements may not be 1, and the elements may not satisfy the inequality abs(a) <= 1. The real and imaginary parts are clipped to the interval [-1, 1] in an attempt to improve on that situation but is not much help in the complex case.

This function accepts but discards arguments *bias* and *ddof*. This is for backwards compatibility with previous versions of this function. These arguments had no effect on the return values of the function and can be safely ignored in this and previous versions of numpy.

`numpy.`**`correlate`**(*a*, *v*, *mode='valid'*)

Cross-correlation of two 1-dimensional sequences.

This function computes the correlation as generally defined in signal processing texts:

```
c_{av}[k] = sum_n a[n+k] * conj(v[n])
```

with a and v sequences being zero-padded where necessary and conj being the conjugate.

**Parameters**

**a, v**

[array_like] Input sequences.

**mode**

[{'valid', 'same', 'full'}, optional] Refer to the `convolve` docstring. Note that the default is 'valid', unlike `convolve`, which uses 'full'.

**old_behavior**

[bool] *old_behavior* was removed in NumPy 1.10. If you need the old behavior, use *multiarray.correlate*.

**Returns**

**out**

[ndarray] Discrete cross-correlation of *a* and *v*.

**See also:**

**`convolve`**

Discrete, linear convolution of two one-dimensional sequences.

**`multiarray.correlate`**

Old, no conjugate, version of correlate.

## Notes

The definition of correlation above is not unique and sometimes correlation may be defined differently. Another common definition is:

```
c'_{av}[k] = sum_n a[n] conj(v[n+k])
```

which is related to `c_{av}[k]` by `c'_{av}[k] = c_{av}[-k]`.

## Examples

```
>>> np.correlate([1, 2, 3], [0, 1, 0.5])
array([3.5])
>>> np.correlate([1, 2, 3], [0, 1, 0.5], "same")
array([2. ,  3.5,  3. ])
>>> np.correlate([1, 2, 3], [0, 1, 0.5], "full")
array([0.5,  2. ,  3.5,  3. ,  0. ])
```

Using complex sequences:

```
>>> np.correlate([1+1j, 2, 3-1j], [0, 1, 0.5j], 'full')
array([ 0.5-0.5j,  1.0+0.j ,  1.5-1.5j,  3.0-1.j ,  0.0+0.j ])
```

Note that you get the time reversed, complex conjugated result when the two input sequences change places, i.e., `c_{va}[k] = c^{*}_{av}[-k]`:

```
>>> np.correlate([0, 1, 0.5j], [1+1j, 2, 3-1j], 'full')
array([ 0.0+0.j ,  3.0+1.j ,  1.5+1.5j,  1.0+0.j ,  0.5+0.5j])
```

`numpy.`**`cov`**(*m*, *y=None*, *rowvar=True*, *bias=False*, *ddof=None*, *fweights=None*, *aweights=None*)

Estimate a covariance matrix, given data and weights.

Covariance indicates the level to which two variables vary together. If we examine N-dimensional samples, $X = [x_1, x_2, ...x_N]^T$, then the covariance matrix element $C_{ij}$ is the covariance of $x_i$ and $x_j$. The element $C_{ii}$ is the variance of $x_i$.

See the notes for an outline of the algorithm.

**Parameters**

**m**

[array_like] A 1-D or 2-D array containing multiple variables and observations. Each row of *m* represents a variable, and each column a single observation of all those variables. Also see *rowvar* below.

**y**

[array_like, optional] An additional set of variables and observations. *y* has the same form as that of *m*.

**rowvar**

[bool, optional] If *rowvar* is True (default), then each row represents a variable, with observations in the columns. Otherwise, the relationship is transposed: each column represents a variable, while the rows contain observations.

**bias**

[bool, optional] Default normalization (False) is by `(N - 1)`, where `N` is the number of observations given (unbiased estimate). If *bias* is True, then normalization is by `N`. These values can be overridden by using the keyword `ddof` in numpy versions >= 1.5.

**ddof**

[int, optional] If not `None` the default value implied by *bias* is overridden. Note that `ddof=1` will return the unbiased estimate, even if both *fweights* and *aweights* are specified, and `ddof=0` will return the simple average. See the notes for the details. The default value is `None`.

New in version 1.5.

**fweights**

[array_like, int, optional] 1-D array of integer frequency weights; the number of times each observation vector should be repeated.

New in version 1.10.

**aweights**

[array_like, optional] 1-D array of observation vector weights. These relative weights are typically large for observations considered "important" and smaller for observations considered less "important". If `ddof=0` the array of weights can be used to assign probabilities to observation vectors.

New in version 1.10.

**Returns**

**out**

[ndarray] The covariance matrix of the variables.

**See also:**

**`corrcoef`**

Normalized covariance matrix

## Notes

Assume that the observations are in the columns of the observation array *m* and let `f = fweights` and `a = aweights` for brevity. The steps to compute the weighted covariance are as follows:

```
>>> m = np.arange(10, dtype=np.float64)
>>> f = np.arange(10) * 2
>>> a = np.arange(10) ** 2.
>>> ddof = 1
>>> w = f * a
>>> v1 = np.sum(w)
>>> v2 = np.sum(w * a)
>>> m -= np.sum(m * w, axis=None, keepdims=True) / v1
>>> cov = np.dot(m * w, m.T) * v1 / (v1**2 - ddof * v2)
```

Note that when `a == 1`, the normalization factor `v1 / (v1**2 - ddof * v2)` goes over to `1 / (np.sum(f) - ddof)` as it should.

### Examples

Consider two variables, $x_0$ and $x_1$, which correlate perfectly, but in opposite directions:

```
>>> x = np.array([[0, 2], [1, 1], [2, 0]]).T
>>> x
array([[0, 1, 2],
       [2, 1, 0]])
```

Note how $x_0$ increases while $x_1$ decreases. The covariance matrix shows this clearly:

```
>>> np.cov(x)
array([[ 1., -1.],
       [-1.,  1.]])
```

Note that element $C_{0,1}$, which shows the correlation between $x_0$ and $x_1$, is negative.

Further, note how *x* and *y* are combined:

```
>>> x = [-2.1, -1,  4.3]
>>> y = [3,  1.1,  0.12]
>>> X = np.stack((x, y), axis=0)
>>> np.cov(X)
array([[11.71      , -4.286     ], # may vary
       [-4.286     ,  2.144133]])
>>> np.cov(x, y)
array([[11.71      , -4.286     ], # may vary
       [-4.286     ,  2.144133]])
>>> np.cov(x)
array(11.71)
```

## 4.27.4 Histograms

| | |
|---|---|
| `histogram`(a[, bins, range, normed, weights, …]) | Compute the histogram of a set of data. |
| `histogram2d`(x, y[, bins, range, normed, …]) | Compute the bi-dimensional histogram of two data samples. |
| `histogramdd`(sample[, bins, range, normed, …]) | Compute the multidimensional histogram of some data. |
| `bincount`(x[, weights, minlength]) | Count number of occurrences of each value in array of non-negative ints. |
| `histogram_bin_edges`(a[, bins, range, weights]) | Function to calculate only the edges of the bins used by the `histogram` function. |
| `digitize`(x, bins[, right]) | Return the indices of the bins to which each value in input array belongs. |

`numpy.`**`histogram`**(*a*, *bins=10*, *range=None*, *normed=None*, *weights=None*, *density=None*)

Compute the histogram of a set of data.

**Parameters**

**a**

[array_like] Input data. The histogram is computed over the flattened array.

**bins**

[int or sequence of scalars or str, optional] If *bins* is an int, it defines the number of equal-width bins in the given range (10, by default). If *bins* is a sequence, it defines a monotonically increasing array of bin edges, including the rightmost edge, allowing for non-uniform bin widths.

New in version 1.11.0.

If *bins* is a string, it defines the method used to calculate the optimal bin width, as defined by `histogram_bin_edges`.

**range**

[(float, float), optional] The lower and upper range of the bins. If not provided, range is simply `(a.min(), a.max())`. Values outside the range are ignored. The first element of the range must be less than or equal to the second. *range* affects the automatic bin computation as well. While bin width is computed to be optimal based on the actual data within *range*, the bin count will fill the entire range including portions containing no data.

**normed**

[bool, optional] Deprecated since version 1.6.0.

This is equivalent to the *density* argument, but produces incorrect results for unequal bin widths. It should not be used.

Changed in version 1.15.0: DeprecationWarnings are actually emitted.

**weights**

[array_like, optional] An array of weights, of the same shape as *a*. Each value in *a* only contributes its associated weight towards the bin count (instead of 1). If *density* is True, the weights are normalized, so that the integral of the density over the range remains 1.

**density**

[bool, optional] If `False`, the result will contain the number of samples in each bin. If `True`, the result is the value of the probability *density* function at the bin, normalized such that the *integral* over the range is 1. Note that the sum of the histogram values will not be equal to 1 unless bins of unity width are chosen; it is not a probability *mass* function.

Overrides the `normed` keyword if given.

**Returns**

**hist**

[array] The values of the histogram. See *density* and *weights* for a description of the possible semantics.

**bin_edges**

[array of dtype float] Return the bin edges `(length(hist)+1)`.

**See also:**

`histogramdd`, `bincount`, `searchsorted`, `digitize`, `histogram_bin_edges`

### Notes

All but the last (righthand-most) bin is half-open. In other words, if *bins* is:

```
[1, 2, 3, 4]
```

then the first bin is `[1, 2)` (including 1, but excluding 2) and the second `[2, 3)`. The last bin, however, is `[3, 4]`, which *includes* 4.

### Examples

```
>>> np.histogram([1, 2, 1], bins=[0, 1, 2, 3])
(array([0, 2, 1]), array([0, 1, 2, 3]))
>>> np.histogram(np.arange(4), bins=np.arange(5), density=True)
(array([0.25, 0.25, 0.25, 0.25]), array([0, 1, 2, 3, 4]))
>>> np.histogram([[1, 2, 1], [1, 0, 1]], bins=[0,1,2,3])
(array([1, 4, 1]), array([0, 1, 2, 3]))
```

```
>>> a = np.arange(5)
>>> hist, bin_edges = np.histogram(a, density=True)
>>> hist
array([0.5, 0. , 0.5, 0. , 0. , 0.5, 0. , 0.5, 0. , 0.5])
>>> hist.sum()
2.4999999999999996
>>> np.sum(hist * np.diff(bin_edges))
1.0
```

New in version 1.11.0.

Automated Bin Selection Methods example, using 2 peak random data with 2000 points:

```
>>> import matplotlib.pyplot as plt
>>> rng = np.random.RandomState(10)  # deterministic random data
>>> a = np.hstack((rng.normal(size=1000),
...                rng.normal(loc=5, scale=2, size=1000)))
>>> _ = plt.hist(a, bins='auto')  # arguments are passed to np.histogram
>>> plt.title("Histogram with 'auto' bins")
Text(0.5, 1.0, "Histogram with 'auto' bins")
>>> plt.show()
```

`numpy.`**`histogram2d`**(*x*, *y*, *bins=10*, *range=None*, *normed=None*, *weights=None*, *density=None*)

Compute the bi-dimensional histogram of two data samples.

**Parameters**

**x**

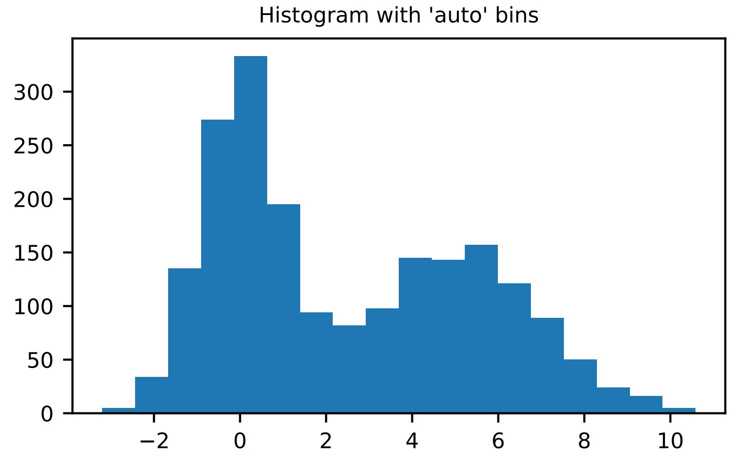
[array_like, shape (N,)] An array containing the x coordinates of the points to be histogrammed.

**y**

[array_like, shape (N,)] An array containing the y coordinates of the points to be histogrammed.

**bins**

[int or array_like or [int, int] or [array, array], optional] The bin specification:



- If int, the number of bins for the two dimensions (nx=ny=bins).
- If array_like, the bin edges for the two dimensions (x_edges=y_edges=bins).
- If [int, int], the number of bins in each dimension (nx, ny = bins).
- If [array, array], the bin edges in each dimension (x_edges, y_edges = bins).
- A combination [int, array] or [array, int], where int is the number of bins and array is the bin edges.

**range**

[array_like, shape(2,2), optional] The leftmost and rightmost edges of the bins along each dimension (if not specified explicitly in the *bins* parameters): `[[xmin, xmax], [ymin, ymax]]`. All values outside of this range will be considered outliers and not tallied in the histogram.

**density**

[bool, optional] If False, the default, returns the number of samples in each bin. If True, returns the probability *density* function at the bin, `bin_count / sample_count / bin_area`.

**normed**

[bool, optional] An alias for the density argument that behaves identically. To avoid confusion with the broken normed argument to `histogram`, *density* should be preferred.

**weights**

[array_like, shape(N,), optional] An array of values `w_i` weighing each sample `(x_i, y_i)`. Weights are normalized to 1 if *normed* is True. If *normed* is False, the values of the returned histogram are equal to the sum of the weights belonging to the samples falling into each bin.

**Returns**

**H**

[ndarray, shape(nx, ny)] The bi-dimensional histogram of samples *x* and *y*. Values in *x* are histogrammed along the first dimension and values in *y* are histogrammed along the second dimension.

**xedges**

[ndarray, shape(nx+1,)] The bin edges along the first dimension.

**yedges**

[ndarray, shape(ny+1,)] The bin edges along the second dimension.

**See also:**

***`histogram`***

1D histogram

***`histogramdd`***

Multidimensional histogram

## Notes

When *normed* is True, then the returned histogram is the sample density, defined such that the sum over bins of the product `bin_value * bin_area` is 1.

Please note that the histogram does not follow the Cartesian convention where *x* values are on the abscissa and *y* values on the ordinate axis. Rather, *x* is histogrammed along the first dimension of the array (vertical), and *y* along the second dimension of the array (horizontal). This ensures compatibility with `histogramdd`.

## Examples

```
>>> from matplotlib.image import NonUniformImage
>>> import matplotlib.pyplot as plt
```

Construct a 2-D histogram with variable bin width. First define the bin edges:

```
>>> xedges = [0, 1, 3, 5]
>>> yedges = [0, 2, 3, 4, 6]
```

Next we create a histogram H with random bin content:

```
>>> x = np.random.normal(2, 1, 100)
>>> y = np.random.normal(1, 1, 100)
>>> H, xedges, yedges = np.histogram2d(x, y, bins=(xedges, yedges))
>>> H = H.T  # Let each row list bins with common y range.
```

`imshow` can only display square bins:

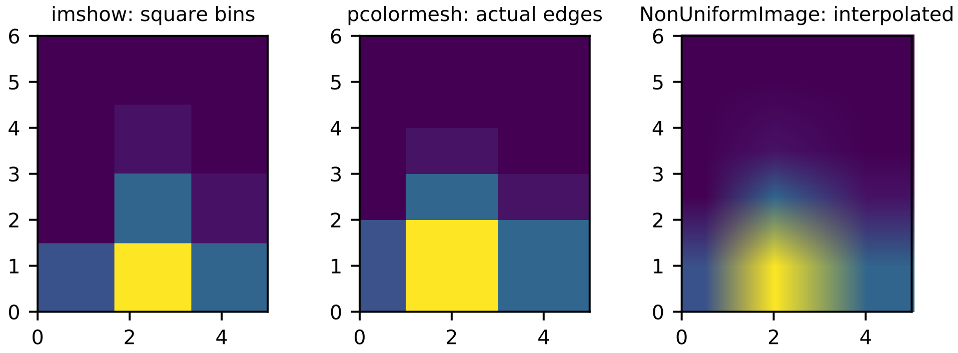
```
>>> fig = plt.figure(figsize=(7, 3))
>>> ax = fig.add_subplot(131, title='imshow: square bins')
>>> plt.imshow(H, interpolation='nearest', origin='low',
...         extent=[xedges[0], xedges[-1], yedges[0], yedges[-1]])
<matplotlib.image.AxesImage object at 0x...>
```

`pcolormesh` can display actual edges:

```
>>> ax = fig.add_subplot(132, title='pcolormesh: actual edges',
...         aspect='equal')
>>> X, Y = np.meshgrid(xedges, yedges)
>>> ax.pcolormesh(X, Y, H)
<matplotlib.collections.QuadMesh object at 0x...>
```

`NonUniformImage` can be used to display actual bin edges with interpolation:

```
>>> ax = fig.add_subplot(133, title='NonUniformImage: interpolated',
...         aspect='equal', xlim=xedges[[0, -1]], ylim=yedges[[0, -1]])
>>> im = NonUniformImage(ax, interpolation='bilinear')
>>> xcenters = (xedges[:-1] + xedges[1:]) / 2
>>> ycenters = (yedges[:-1] + yedges[1:]) / 2
>>> im.set_data(xcenters, ycenters, H)
>>> ax.images.append(im)
>>> plt.show()
```



`numpy.histogramdd`(*sample*, *bins=10*, *range=None*, *normed=None*, *weights=None*, *density=None*)

Compute the multidimensional histogram of some data.

**Parameters**

**sample**

[(N, D) array, or (D, N) array_like] The data to be histogrammed.

Note the unusual interpretation of sample when an array_like:

- When an array, each row is a coordinate in a D-dimensional space - such as `histogramdd(np.array([p1, p2, p3]))`.
- When an array_like, each element is the list of values for single coordinate - such as `histogramdd((X, Y, Z))`.

The first form should be preferred.

**bins**

[sequence or int, optional] The bin specification:

- A sequence of arrays describing the monotonically increasing bin edges along each dimension.
- The number of bins for each dimension (nx, ny, … =bins)
- The number of bins for all dimensions (nx=ny=…=bins).

**range**

[sequence, optional] A sequence of length D, each an optional (lower, upper) tuple giving the outer bin edges to be used if the edges are not given explicitly in *bins*. An entry of None in the sequence results in the minimum and maximum values being used for the corresponding dimension. The default, None, is equivalent to passing a tuple of D None values.

**density**

[bool, optional] If False, the default, returns the number of samples in each bin. If True, returns the probability *density* function at the bin, `bin_count / sample_count / bin_volume`.

**normed**

[bool, optional] An alias for the density argument that behaves identically. To avoid confusion with the broken normed argument to `histogram`, *density* should be preferred.

**weights**

[(N,) array_like, optional] An array of values *w_i* weighing each sample *(x_i, y_i, z_i, …)*. Weights are normalized to 1 if normed is True. If normed is False, the values of the returned histogram are equal to the sum of the weights belonging to the samples falling into each bin.

**Returns**

**H**

[ndarray] The multidimensional histogram of sample x. See normed and weights for the different possible semantics.

**edges**

[list] A list of D arrays describing the bin edges for each dimension.

**See also:**

**`histogram`**

1-D histogram

**`histogram2d`**

2-D histogram

### Examples

```
>>> r = np.random.randn(100,3)
>>> H, edges = np.histogramdd(r, bins = (5, 8, 4))
>>> H.shape, edges[0].size, edges[1].size, edges[2].size
((5, 8, 4), 6, 9, 5)
```

`numpy.`**`bincount`**(*x*, *weights=None*, *minlength=0*)

Count number of occurrences of each value in array of non-negative ints.

The number of bins (of size 1) is one larger than the largest value in *x*. If *minlength* is specified, there will be at least this number of bins in the output array (though it will be longer if necessary, depending on the contents of *x*). Each bin gives the number of occurrences of its index value in *x*. If *weights* is specified the input array is weighted by it, i.e. if a value `n` is found at position `i`, `out[n] += weight[i]` instead of `out[n] += 1`.

**Parameters**

**x**

[array_like, 1 dimension, nonnegative ints] Input array.

**weights**

[array_like, optional] Weights, array of the same shape as *x*.

**minlength**

[int, optional] A minimum number of bins for the output array.

New in version 1.6.0.

**Returns**

**out**

[ndarray of ints] The result of binning the input array. The length of *out* is equal to `np.amax(x)+1`.

**Raises**

**ValueError**

If the input is not 1-dimensional, or contains elements with negative values, or if *minlength* is negative.

**TypeError**

If the type of the input is float or complex.

**See also:**

*`histogram`*, *`digitize`*, *`unique`*

### Examples

```
>>> np.bincount(np.arange(5))
array([1, 1, 1, 1, 1])
>>> np.bincount(np.array([0, 1, 1, 3, 2, 1, 7]))
array([1, 3, 1, 1, 0, 0, 0, 1])
```

```
>>> x = np.array([0, 1, 1, 3, 2, 1, 7, 23])
>>> np.bincount(x).size == np.amax(x)+1
True
```

The input array needs to be of integer dtype, otherwise a TypeError is raised:

```
>>> np.bincount(np.arange(5, dtype=float))
Traceback (most recent call last):
  ...
TypeError: Cannot cast array data from dtype('float64') to dtype('int64')
according to the rule 'safe'
```

A possible use of `bincount` is to perform sums over variable-size chunks of an array, using the `weights` keyword.

```
>>> w = np.array([0.3, 0.5, 0.2, 0.7, 1., -0.6]) # weights
>>> x = np.array([0, 1, 1, 2, 2, 2])
>>> np.bincount(x,  weights=w)
array([ 0.3,  0.7,  1.1])
```

`numpy.`**`histogram_bin_edges`**(*a*, *bins=10*, *range=None*, *weights=None*)

Function to calculate only the edges of the bins used by the *`histogram`* function.

**Parameters**

**a**

[array_like] Input data. The histogram is computed over the flattened array.

**bins**

[int or sequence of scalars or str, optional] If *bins* is an int, it defines the number of equal-width bins in the given range (10, by default). If *bins* is a sequence, it defines the bin edges, including the rightmost edge, allowing for non-uniform bin widths.

If *bins* is a string from the list below, *`histogram_bin_edges`* will use the method chosen to calculate the optimal bin width and consequently the number of bins (see *Notes* for more detail on the estimators) from the data that falls within the requested range. While the bin width will be optimal for the actual data in the range, the number of bins will be computed to fill the entire range, including the empty portions. For visualisation, using the 'auto' option is suggested. Weighted data is not supported for automated bin size selection.

**'auto'**

Maximum of the 'sturges' and 'fd' estimators. Provides good all around performance.

**'fd' (Freedman Diaconis Estimator)**

Robust (resilient to outliers) estimator that takes into account data variability and data size.

**'doane'**

An improved version of Sturges' estimator that works better with non-normal datasets.

**'scott'**

Less robust estimator that that takes into account data variability and data size.

**'stone'**

Estimator based on leave-one-out cross-validation estimate of the integrated squared error. Can be regarded as a generalization of Scott's rule.

**'rice'**

Estimator does not take variability into account, only data size. Commonly overestimates number of bins required.

**'sturges'**

R's default method, only accounts for data size. Only optimal for gaussian data and underestimates number of bins for large non-gaussian datasets.

**'sqrt'**

Square root (of data size) estimator, used by Excel and other programs for its speed and simplicity.

**range**

[(float, float), optional] The lower and upper range of the bins. If not provided, range is simply `(a.min(), a.max())`. Values outside the range are ignored. The first element of the range must be less than or equal to the second. *range* affects the automatic bin computation as well. While bin width is computed to be optimal based on the actual data within *range*, the bin count will fill the entire range including portions containing no data.

**weights**

[array_like, optional] An array of weights, of the same shape as *a*. Each value in *a* only contributes its associated weight towards the bin count (instead of 1). This is currently not used by any of the bin estimators, but may be in the future.

**Returns**

**bin_edges**

[array of dtype float] The edges to pass into `histogram`

**See also:**

`histogram`

## Notes

The methods to estimate the optimal number of bins are well founded in literature, and are inspired by the choices R provides for histogram visualisation. Note that having the number of bins proportional to $n^{1/3}$ is asymptotically optimal, which is why it appears in most estimators. These are simply plug-in methods that give good starting points for number of bins. In the equations below, $h$ is the binwidth and $n_h$ is the number of bins. All estimators that compute bin counts are recast to bin width using the `ptp` of the data. The final bin count is obtained from `np.round(np.ceil(range / h))`.

**'auto' (maximum of the 'sturges' and 'fd' estimators)**

A compromise to get a good value. For small datasets the Sturges value will usually be chosen, while larger datasets will usually default to FD. Avoids the overly conservative behaviour of FD and Sturges for small and large datasets respectively. Switchover point is usually $a.size \approx 1000$.

**'fd' (Freedman Diaconis Estimator)**

$$h = 2\frac{IQR}{n^{1/3}}$$

The binwidth is proportional to the interquartile range (IQR) and inversely proportional to cube root of a.size. Can be too conservative for small datasets, but is quite good for large datasets. The IQR is very robust to outliers.

**'scott'**

$$h = \sigma\sqrt[3]{\frac{24 * \sqrt{\pi}}{n}}$$

The binwidth is proportional to the standard deviation of the data and inversely proportional to cube root of `x.size`. Can be too conservative for small datasets, but is quite good for large datasets. The standard deviation is not very robust to outliers. Values are very similar to the Freedman-Diaconis estimator in the absence of outliers.

**'rice'**

$$n_h = 2n^{1/3}$$

The number of bins is only proportional to cube root of `a.size`. It tends to overestimate the number of bins and it does not take into account data variability.

**'sturges'**

$$n_h = \log_2 n + 1$$

The number of bins is the base 2 log of `a.size`. This estimator assumes normality of data and is too conservative for larger, non-normal datasets. This is the default method in R's `hist` method.

**'doane'**

$$\begin{aligned} n_h &= 1 + \log_2(n) + \log_2(1 + \frac{|g_1|}{\sigma_{g_1}}) \\ g_1 &= mean[(\frac{x - \mu}{\sigma})^3] \\ \sigma_{g_1} &= \sqrt{\frac{6(n - 2)}{(n + 1)(n + 3)}} \end{aligned}$$

An improved version of Sturges' formula that produces better estimates for non-normal datasets. This estimator attempts to account for the skew of the data.

**'sqrt'**

$$n_h = \sqrt{n}$$

The simplest and fastest estimator. Only takes into account the data size.

### Examples

```
>>> arr = np.array([0, 0, 0, 1, 2, 3, 3, 4, 5])
>>> np.histogram_bin_edges(arr, bins='auto', range=(0, 1))
array([0.  , 0.25, 0.5 , 0.75, 1.  ])
>>> np.histogram_bin_edges(arr, bins=2)
array([0. , 2.5, 5. ])
```

For consistency with histogram, an array of pre-computed bins is passed through unmodified:

```
>>> np.histogram_bin_edges(arr, [1, 2])
array([1, 2])
```

This function allows one set of bins to be computed, and reused across multiple histograms:

```
>>> shared_bins = np.histogram_bin_edges(arr, bins='auto')
>>> shared_bins
array([0., 1., 2., 3., 4., 5.])
```

```
>>> group_id = np.array([0, 1, 1, 0, 1, 1, 0, 1, 1])
>>> hist_0, _ = np.histogram(arr[group_id == 0], bins=shared_bins)
>>> hist_1, _ = np.histogram(arr[group_id == 1], bins=shared_bins)
```

```
>>> hist_0; hist_1
array([1, 1, 0, 1, 0])
array([2, 0, 1, 1, 2])
```

Which gives more easily comparable results than using separate bins for each histogram:

```
>>> hist_0, bins_0 = np.histogram(arr[group_id == 0], bins='auto')
>>> hist_1, bins_1 = np.histogram(arr[group_id == 1], bins='auto')
>>> hist_0; hist_1
array([1, 1, 1])
array([2, 1, 1, 2])
>>> bins_0; bins_1
array([0., 1., 2., 3.])
array([0.  , 1.25, 2.5 , 3.75, 5.  ])
```

`numpy.`**`digitize`**(*x*, *bins*, *right=False*)

Return the indices of the bins to which each value in input array belongs.

| *right* | order of bins | returned index *i* satisfies |
|---|---|---|
| `False` | increasing | `bins[i-1] <= x < bins[i]` |
| `True` | increasing | `bins[i-1] < x <= bins[i]` |
| `False` | decreasing | `bins[i-1] > x >= bins[i]` |
| `True` | decreasing | `bins[i-1] >= x > bins[i]` |

If values in *x* are beyond the bounds of *bins*, 0 or `len(bins)` is returned as appropriate.

**Parameters**

**x**

[array_like] Input array to be binned. Prior to NumPy 1.10.0, this array had to be 1-dimensional, but can now have any shape.

**bins**

[array_like] Array of bins. It has to be 1-dimensional and monotonic.

**right**

[bool, optional] Indicating whether the intervals include the right or the left bin edge. Default behavior is (right==False) indicating that the interval does not include the right edge. The left bin end is open in this case, i.e., bins[i-1] <= x < bins[i] is the default behavior for monotonically increasing bins.

**Returns**

**indices**

[ndarray of ints] Output array of indices, of same shape as *x*.

**Raises**

**ValueError**

If *bins* is not monotonic.

**TypeError**

If the type of the input is complex.

**See also:**

`bincount`, `histogram`, `unique`, `searchsorted`

## Notes

If values in *x* are such that they fall outside the bin range, attempting to index *bins* with the indices that `digitize` returns will result in an IndexError.

New in version 1.10.0.

*np.digitize* is implemented in terms of *np.searchsorted*. This means that a binary search is used to bin the values, which scales much better for larger number of bins than the previous linear search. It also removes the requirement for the input array to be 1-dimensional.

For monotonically _increasing_ *bins*, the following are equivalent:

```
np.digitize(x, bins, right=True)
np.searchsorted(bins, x, side='left')
```

Note that as the order of the arguments are reversed, the side must be too. The `searchsorted` call is marginally faster, as it does not do any monotonicity checks. Perhaps more importantly, it supports all dtypes.

### Examples

```
>>> x = np.array([0.2, 6.4, 3.0, 1.6])
>>> bins = np.array([0.0, 1.0, 2.5, 4.0, 10.0])
>>> inds = np.digitize(x, bins)
>>> inds
array([1, 4, 3, 2])
>>> for n in range(x.size):
...   print(bins[inds[n]-1], "<=", x[n], "<", bins[inds[n]])
...
0.0 <= 0.2 < 1.0
4.0 <= 6.4 < 10.0
2.5 <= 3.0 < 4.0
1.0 <= 1.6 < 2.5
```

```
>>> x = np.array([1.2, 10.0, 12.4, 15.5, 20.])
>>> bins = np.array([0, 5, 10, 15, 20])
>>> np.digitize(x,bins,right=True)
array([1, 2, 3, 4, 4])
>>> np.digitize(x,bins,right=False)
array([1, 3, 3, 4, 5])
```

## 4.28 Test Support (`numpy.testing`)

Common test support for all numpy test scripts.

This single module should provide all the common functionality for numpy tests in a single location, so that test scripts can just import it and work right away. For background, see the *Testing Guidelines*

### 4.28.1 Asserts

| | |
|---|---|
| *assert_almost_equal*(actual, desired[, …]) | Raises an AssertionError if two items are not equal up to desired precision. |
| *assert_approx_equal*(actual, desired[, …]) | Raises an AssertionError if two items are not equal up to significant digits. |
| *assert_array_almost_equal*(x, y[, decimal, …]) | Raises an AssertionError if two objects are not equal up to desired precision. |
| *assert_allclose*(actual, desired[, rtol, …]) | Raises an AssertionError if two objects are not equal up to desired tolerance. |
| *assert_array_almost_equal_nulp*(x, y[, nulp]) | Compare two arrays relatively to their spacing. |
| *assert_array_max_ulp*(a, b[, maxulp, dtype]) | Check that all items of arrays differ in at most N Units in the Last Place. |
| *assert_array_equal*(x, y[, err_msg, verbose]) | Raises an AssertionError if two array_like objects are not equal. |
| *assert_array_less*(x, y[, err_msg, verbose]) | Raises an AssertionError if two array_like objects are not ordered by less than. |
| *assert_equal*(actual, desired[, err_msg, verbose]) | Raises an AssertionError if two objects are not equal. |

Continued on next page

Table 176 – continued from previous page

| | |
|---|---|
| *assert_raises*(exception_class, callable, …) | Fail unless an exception of class exception_class is thrown by callable when invoked with arguments args and keyword arguments kwargs. |
| *assert_raises_regex*(exception_class, …) | Fail unless an exception of class exception_class and with message that matches expected_regexp is thrown by callable when invoked with arguments args and keyword arguments kwargs. |
| *assert_warns*(warning_class, *args, **kwargs) | Fail unless the given callable throws the specified warning. |
| *assert_string_equal*(actual, desired) | Test if two strings are equal. |

`numpy.testing.`**`assert_almost_equal`**(*actual*, *desired*, *decimal=7*, *err_msg=''*, *verbose=True*)

Raises an AssertionError if two items are not equal up to desired precision.

---

**Note:** It is recommended to use one of *assert_allclose*, *assert_array_almost_equal_nulp* or *assert_array_max_ulp* instead of this function for more consistent floating point comparisons.

---

The test verifies that the elements of `actual` and `desired` satisfy.

```
abs(desired-actual) < 1.5 * 10**(-decimal)
```

That is a looser test than originally documented, but agrees with what the actual implementation in *assert_array_almost_equal* did up to rounding vagaries. An exception is raised at conflicting values. For ndarrays this delegates to assert_array_almost_equal

**Parameters**

**actual**

[array_like] The object to check.

**desired**

[array_like] The expected object.

**decimal**

[int, optional] Desired precision, default is 7.

**err_msg**

[str, optional] The error message to be printed in case of failure.

**verbose**

[bool, optional] If True, the conflicting values are appended to the error message.

**Raises**

**AssertionError**

If actual and desired are not equal up to specified precision.

**See also:**

***assert_allclose***

Compare two array_like objects for equality with desired relative and/or absolute precision.

*assert_array_almost_equal_nulp*, *assert_array_max_ulp*, *assert_equal*

### Examples

```
>>> import numpy.testing as npt
>>> npt.assert_almost_equal(2.3333333333333, 2.33333334)
>>> npt.assert_almost_equal(2.3333333333333, 2.33333334, decimal=10)
Traceback (most recent call last):
    ...
AssertionError:
Arrays are not almost equal to 10 decimals
 ACTUAL: 2.3333333333333
 DESIRED: 2.33333334
```

```
>>> npt.assert_almost_equal(np.array([1.0,2.3333333333333]),
...                         np.array([1.0,2.33333334]), decimal=9)
Traceback (most recent call last):
    ...
AssertionError:
Arrays are not almost equal to 9 decimals

Mismatched elements: 1 / 2 (50%)
Max absolute difference: 6.66669964e-09
Max relative difference: 2.85715698e-09
 x: array([1.         , 2.333333333])
 y: array([1.        , 2.33333334])
```

`numpy.testing.`**`assert_approx_equal`**(*actual*, *desired*, *significant=7*, *err_msg=”*, *verbose=True*)

Raises an AssertionError if two items are not equal up to significant digits.

---

**Note:** It is recommended to use one of `assert_allclose`, `assert_array_almost_equal_nulp` or `assert_array_max_ulp` instead of this function for more consistent floating point comparisons.

---

Given two numbers, check that they are approximately equal. Approximately equal is defined as the number of significant digits that agree.

**Parameters**

**actual**

[scalar] The object to check.

**desired**

[scalar] The expected object.

**significant**

[int, optional] Desired precision, default is 7.

**err_msg**

[str, optional] The error message to be printed in case of failure.

**verbose**

[bool, optional] If True, the conflicting values are appended to the error message.

**Raises**

**AssertionError**

If actual and desired are not equal up to specified precision.

**See also:**

***assert_allclose***

Compare two array_like objects for equality with desired relative and/or absolute precision.

*assert_array_almost_equal_nulp*, *assert_array_max_ulp*, *assert_equal*

### Examples

```
>>> np.testing.assert_approx_equal(0.12345677777777e-20, 0.1234567e-20)
>>> np.testing.assert_approx_equal(0.12345670e-20, 0.12345671e-20,
...                                significant=8)
>>> np.testing.assert_approx_equal(0.12345670e-20, 0.12345672e-20,
...                                significant=8)
Traceback (most recent call last):
    ...
AssertionError:
Items are not equal to 8 significant digits:
 ACTUAL: 1.234567e-21
 DESIRED: 1.2345672e-21
```

the evaluated condition that raises the exception is

```
>>> abs(0.12345670e-20/1e-21 - 0.12345672e-20/1e-21) >= 10**-(8-1)
True
```

`numpy.testing.`**`assert_array_almost_equal`**(*x*, *y*, *decimal=6*, *err_msg=''*, *verbose=True*)

Raises an AssertionError if two objects are not equal up to desired precision.

**Note:** It is recommended to use one of *assert_allclose*, *assert_array_almost_equal_nulp* or *assert_array_max_ulp* instead of this function for more consistent floating point comparisons.

The test verifies identical shapes and that the elements of `actual` and `desired` satisfy.

```
abs(desired-actual) < 1.5 * 10**(-decimal)
```

That is a looser test than originally documented, but agrees with what the actual implementation did up to rounding vagaries. An exception is raised at shape mismatch or conflicting values. In contrast to the standard usage in numpy, NaNs are compared like numbers, no assertion is raised if both objects have NaNs in the same positions.

**Parameters**

**x**

[array_like] The actual object to check.

**y**

[array_like] The desired, expected object.

**decimal**

[int, optional] Desired precision, default is 6.

**err_msg**

[str, optional] The error message to be printed in case of failure.

**verbose**

[bool, optional] If True, the conflicting values are appended to the error message.

**Raises**

**AssertionError**

If actual and desired are not equal up to specified precision.

**See also:**

***assert_allclose***

Compare two array_like objects for equality with desired relative and/or absolute precision.

*assert_array_almost_equal_nulp*, *assert_array_max_ulp*, *assert_equal*

## Examples

the first assert does not raise an exception

```
>>> np.testing.assert_array_almost_equal([1.0,2.333,np.nan],
...                                      [1.0,2.333,np.nan])
```

```
>>> np.testing.assert_array_almost_equal([1.0,2.33333,np.nan],
...                                      [1.0,2.33339,np.nan], decimal=5)
Traceback (most recent call last):
    ...
AssertionError:
Arrays are not almost equal to 5 decimals

Mismatched elements: 1 / 3 (33.3%)
Max absolute difference: 6.e-05
Max relative difference: 2.57136612e-05
 x: array([1.     , 2.33333,     nan])
 y: array([1.     , 2.33339,     nan])
```

```
>>> np.testing.assert_array_almost_equal([1.0,2.33333,np.nan],
...                                      [1.0,2.33333, 5], decimal=5)
Traceback (most recent call last):
    ...
AssertionError:
Arrays are not almost equal to 5 decimals

x and y nan location mismatch:
 x: array([1.     , 2.33333,     nan])
 y: array([1.     , 2.33333, 5.     ])
```

`numpy.testing.`**`assert_allclose`**(*actual*, *desired*, *rtol=1e-07*, *atol=0*, *equal_nan=True*, *err_msg=''*, *verbose=True*)

Raises an AssertionError if two objects are not equal up to desired tolerance.

The test is equivalent to `allclose(actual, desired, rtol, atol)` (note that `allclose` has different default values). It compares the difference between *actual* and *desired* to `atol + rtol * abs(desired)`.

New in version 1.5.0.

**Parameters**

**actual**

[array_like] Array obtained.

**desired**

[array_like] Array desired.

**rtol**

[float, optional] Relative tolerance.

**atol**

[float, optional] Absolute tolerance.

**equal_nan**

[bool, optional.] If True, NaNs will compare equal.

**err_msg**

[str, optional] The error message to be printed in case of failure.

**verbose**

[bool, optional] If True, the conflicting values are appended to the error message.

**Raises**

**AssertionError**

If actual and desired are not equal up to specified precision.

**See also:**

*`assert_array_almost_equal_nulp`*, *`assert_array_max_ulp`*

## Examples

```
>>> x = [1e-5, 1e-3, 1e-1]
>>> y = np.arccos(np.cos(x))
>>> np.testing.assert_allclose(x, y, rtol=1e-5, atol=0)
```

`numpy.testing.`**`assert_array_almost_equal_nulp`**(*x*, *y*, *nulp=1*)

Compare two arrays relatively to their spacing.

This is a relatively robust method to compare two arrays whose amplitude is variable.

**Parameters**

**x, y**

[array_like] Input arrays.

**nulp**

[int, optional] The maximum number of unit in the last place for tolerance (see Notes). Default is 1.

**Returns**

**None**

**Raises**

**AssertionError**

If the spacing between *x* and *y* for one or more elements is larger than *nulp*.

**See also:**

***assert_array_max_ulp***

Check that all items of arrays differ in at most N Units in the Last Place.

**`spacing`**

Return the distance between x and the nearest adjacent number.

## Notes

An assertion is raised if the following condition is not met:

```
abs(x - y) <= nulps * spacing(maximum(abs(x), abs(y)))
```

## Examples

```
>>> x = np.array([1., 1e-10, 1e-20])
>>> eps = np.finfo(x.dtype).eps
>>> np.testing.assert_array_almost_equal_nulp(x, x*eps/2 + x)
```

```
>>> np.testing.assert_array_almost_equal_nulp(x, x*eps + x)
Traceback (most recent call last):
  ...
AssertionError: X and Y are not equal to 1 ULP (max is 2)
```

numpy.testing.**assert_array_max_ulp**(*a*, *b*, *maxulp=1*, *dtype=None*)

Check that all items of arrays differ in at most N Units in the Last Place.

**Parameters**

**a, b**

[array_like] Input arrays to be compared.

**maxulp**

[int, optional] The maximum number of units in the last place that elements of *a* and *b* can differ. Default is 1.

**dtype**

[dtype, optional] Data-type to convert *a* and *b* to if given. Default is None.

**Returns**

**ret**

[ndarray] Array containing number of representable floating point numbers between items in *a* and *b*.

**Raises**

**AssertionError**

If one or more elements differ by more than *maxulp*.

**See also:**

***assert_array_almost_equal_nulp***

Compare two arrays relatively to their spacing.

### Notes

For computing the ULP difference, this API does not differentiate between various representations of NAN (ULP difference between 0x7fc00000 and 0xffc00000 is zero).

### Examples

```
>>> a = np.linspace(0., 1., 100)
>>> res = np.testing.assert_array_max_ulp(a, np.arcsin(np.sin(a)))
```

`numpy.testing.`**`assert_array_equal`**(*x*, *y*, *err_msg=''*, *verbose=True*)

Raises an AssertionError if two array_like objects are not equal.

Given two array_like objects, check that the shape is equal and all elements of these objects are equal (but see the Notes for the special handling of a scalar). An exception is raised at shape mismatch or conflicting values. In contrast to the standard usage in numpy, NaNs are compared like numbers, no assertion is raised if both objects have NaNs in the same positions.

The usual caution for verifying equality with floating point numbers is advised.

**Parameters**

**x**

[array_like] The actual object to check.

**y**

[array_like] The desired, expected object.

**err_msg**

[str, optional] The error message to be printed in case of failure.

**verbose**

[bool, optional] If True, the conflicting values are appended to the error message.

**Raises**

**AssertionError**

If actual and desired objects are not equal.

**See also:**

***`assert_allclose`***

Compare two array_like objects for equality with desired relative and/or absolute precision.

`assert_array_almost_equal_nulp`, `assert_array_max_ulp`, `assert_equal`

## Notes

When one of *x* and *y* is a scalar and the other is array_like, the function checks that each element of the array_like object is equal to the scalar.

## Examples

The first assert does not raise an exception:

```
>>> np.testing.assert_array_equal([1.0,2.33333,np.nan],
...                               [np.exp(0),2.33333, np.nan])
```

Assert fails with numerical imprecision with floats:

```
>>> np.testing.assert_array_equal([1.0,np.pi,np.nan],
...                               [1, np.sqrt(np.pi)**2, np.nan])
Traceback (most recent call last):
    ...
AssertionError:
Arrays are not equal

Mismatched elements: 1 / 3 (33.3%)
Max absolute difference: 4.4408921e-16
Max relative difference: 1.41357986e-16
 x: array([1.      , 3.141593,      nan])
 y: array([1.      , 3.141593,      nan])
```

Use `assert_allclose` or one of the nulp (number of floating point values) functions for these cases instead:

```
>>> np.testing.assert_allclose([1.0,np.pi,np.nan],
...                            [1, np.sqrt(np.pi)**2, np.nan],
...                            rtol=1e-10, atol=0)
```

As mentioned in the Notes section, `assert_array_equal` has special handling for scalars. Here the test checks that each value in *x* is 3:

```
>>> x = np.full((2, 5), fill_value=3)
>>> np.testing.assert_array_equal(x, 3)
```

`numpy.testing.`**`assert_array_less`**(*x*, *y*, *err_msg=''*, *verbose=True*)

Raises an AssertionError if two array_like objects are not ordered by less than.

Given two array_like objects, check that the shape is equal and all elements of the first object are strictly smaller than those of the second object. An exception is raised at shape mismatch or incorrectly ordered values. Shape mismatch does not raise if an object has zero dimension. In contrast to the standard usage in numpy, NaNs are compared, no assertion is raised if both objects have NaNs in the same positions.

**Parameters**

**x**

[array_like] The smaller object to check.

**y**

[array_like] The larger object to compare.

**err_msg**

[string] The error message to be printed in case of failure.

**verbose**

[bool] If True, the conflicting values are appended to the error message.

**Raises**

**AssertionError**

If actual and desired objects are not equal.

**See also:**

***`assert_array_equal`***

tests objects for equality

***`assert_array_almost_equal`***

test objects for equality up to precision

## Examples

```
>>> np.testing.assert_array_less([1.0, 1.0, np.nan], [1.1, 2.0, np.nan])
>>> np.testing.assert_array_less([1.0, 1.0, np.nan], [1, 2.0, np.nan])
Traceback (most recent call last):
    ...
AssertionError:
Arrays are not less-ordered

Mismatched elements: 1 / 3 (33.3%)
Max absolute difference: 1.
Max relative difference: 0.5
 x: array([ 1.,  1., nan])
 y: array([ 1.,  2., nan])
```

```
>>> np.testing.assert_array_less([1.0, 4.0], 3)
Traceback (most recent call last):
    ...
AssertionError:
Arrays are not less-ordered

Mismatched elements: 1 / 2 (50%)
Max absolute difference: 2.
Max relative difference: 0.66666667
 x: array([1., 4.])
 y: array(3)
```

```
>>> np.testing.assert_array_less([1.0, 2.0, 3.0], [4])
Traceback (most recent call last):
    ...
AssertionError:
Arrays are not less-ordered

(shapes (3,), (1,) mismatch)
 x: array([1., 2., 3.])
 y: array([4])
```

`numpy.testing.`**`assert_equal`**(*actual*, *desired*, *err_msg=''*, *verbose=True*)

Raises an AssertionError if two objects are not equal.

Given two objects (scalars, lists, tuples, dictionaries or numpy arrays), check that all elements of these objects are equal. An exception is raised at the first conflicting values.

When one of *actual* and *desired* is a scalar and the other is array_like, the function checks that each element of the array_like object is equal to the scalar.

This function handles NaN comparisons as if NaN was a "normal" number. That is, AssertionError is not raised if both objects have NaNs in the same positions. This is in contrast to the IEEE standard on NaNs, which says that NaN compared to anything must return False.

**Parameters**

**actual**

[array_like] The object to check.

**desired**

[array_like] The expected object.

**err_msg**

[str, optional] The error message to be printed in case of failure.

**verbose**

[bool, optional] If True, the conflicting values are appended to the error message.

**Raises**

**AssertionError**

If actual and desired are not equal.

### Examples

```
>>> np.testing.assert_equal([4,5], [4,6])
Traceback (most recent call last):
    ...
AssertionError:
Items are not equal:
item=1
 ACTUAL: 5
 DESIRED: 6
```

The following comparison does not raise an exception. There are NaNs in the inputs, but they are in the same positions.

```
>>> np.testing.assert_equal(np.array([1.0, 2.0, np.nan]), [1, 2, np.nan])
```

`numpy.testing.`**`assert_raises`**(*exception_class*, *callable*, **args*, ***kwargs) assert_raises(exception_class*)

Fail unless an exception of class exception_class is thrown by callable when invoked with arguments args and keyword arguments kwargs. If a different type of exception is thrown, it will not be caught, and the test case will be deemed to have suffered an error, exactly as for an unexpected exception.

Alternatively, *`assert_raises`* can be used as a context manager:

```
>>> from numpy.testing import assert_raises
>>> with assert_raises(ZeroDivisionError):
...     1 / 0
```

is equivalent to

```
>>> def div(x, y):
...     return x / y
>>> assert_raises(ZeroDivisionError, div, 1, 0)
```

`numpy.testing.`**`assert_raises_regex`**(*exception_class*, *expected_regexp*, *callable*, **args*, ***kwargs) assert_raises_regex(exception_class*, *expected_regexp*)

Fail unless an exception of class exception_class and with message that matches expected_regexp is thrown by callable when invoked with arguments args and keyword arguments kwargs.

Alternatively, can be used as a context manager like *`assert_raises`*.

Name of this function adheres to Python 3.2+ reference, but should work in all versions down to 2.6.

### Notes

New in version 1.9.0.

`numpy.testing.`**`assert_warns`**(*warning_class*, **args*, ***kwargs*)

Fail unless the given callable throws the specified warning.

A warning of class warning_class should be thrown by the callable when invoked with arguments args and keyword arguments kwargs. If a different type of warning is thrown, it will not be caught.

If called with all arguments other than the warning class omitted, may be used as a context manager:

**with assert_warns(SomeWarning):**

do_something()

The ability to be used as a context manager is new in NumPy v1.11.0.

New in version 1.4.0.

**Parameters**

**warning_class**

[class] The class defining the warning that *func* is expected to throw.

**func**

[callable] The callable to test.

***args**

[Arguments] Arguments passed to *func*.

****kwargs**

[Kwargs] Keyword arguments passed to *func*.

**Returns**

**The value returned by 'func'.**

`numpy.testing.`**`assert_string_equal`**`(`*actual*, *desired*`)`

Test if two strings are equal.

If the given strings are equal, *`assert_string_equal`* does nothing. If they are not equal, an AssertionError is raised, and the diff between the strings is shown.

**Parameters**

**actual**

[str] The string to test for equality against the expected string.

**desired**

[str] The expected string.

## Examples

```
>>> np.testing.assert_string_equal('abc', 'abc')
>>> np.testing.assert_string_equal('abc', 'abcd')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
...
AssertionError: Differences in strings:
- abc+ abcd?    +
```

## 4.28.2 Decorators

| | |
|---|---|
| `dec.deprecated`([conditional]) | Filter deprecation warnings while running the test suite. |
| `dec.knownfailureif`(fail_condition[, msg]) | Make function raise KnownFailureException exception if given condition is true. |
| `dec.setastest`([tf]) | Signals to nose that this function is or is not a test. |
| `dec.skipif`(skip_condition[, msg]) | Make function raise SkipTest exception if a given condition is true. |
| `dec.slow`(t) | Label a test as 'slow'. |
| `decorate_methods`(cls, decorator[, testmatch]) | Apply a decorator to all methods in a class matching a regular expression. |

`numpy.testing.dec.`**`deprecated`**`(`*conditional=True*`)`

Filter deprecation warnings while running the test suite.

This decorator can be used to filter DeprecationWarning's, to avoid printing them during the test suite run, while checking that the test actually raises a DeprecationWarning.

**Parameters**

**conditional**

[bool or callable, optional] Flag to determine whether to mark test as deprecated or not. If the condition is a callable, it is used at runtime to dynamically make the decision. Default is True.

**Returns**

**decorator**

[function] The `deprecated` decorator itself.

### Notes

New in version 1.4.0.

`numpy.testing.dec.`**`knownfailureif`**`(`*fail_condition*, *msg=None*`)`

Make function raise KnownFailureException exception if given condition is true.

If the condition is a callable, it is used at runtime to dynamically make the decision. This is useful for tests that may require costly imports, to delay the cost until the test suite is actually executed.

**Parameters**

**fail_condition**

[bool or callable] Flag to determine whether to mark the decorated test as a known failure (if True) or not (if False).

**msg**

[str, optional] Message to give on raising a KnownFailureException exception. Default is None.

**Returns**

**decorator**

[function] Decorator, which, when applied to a function, causes KnownFailureException to be raised when *fail_condition* is True, and the function to be called normally otherwise.

## Notes

The decorator itself is decorated with the `nose.tools.make_decorator` function in order to transmit function name, and various other metadata.

`numpy.testing.dec.`**`setastest`**`(`*tf=True*`)`

Signals to nose that this function is or is not a test.

**Parameters**

**tf**

[bool] If True, specifies that the decorated callable is a test. If False, specifies that the decorated callable is not a test. Default is True.

## Notes

This decorator can't use the nose namespace, because it can be called from a non-test module. See also `istest` and `nottest` in `nose.tools`.

## Examples

`setastest` can be used in the following way:

```
from numpy.testing import dec

@dec.setastest(False)
def func_with_test_in_name(arg1, arg2):
    pass
```

`numpy.testing.dec.`**`skipif`**`(`*skip_condition, msg=None*`)`

Make function raise SkipTest exception if a given condition is true.

If the condition is a callable, it is used at runtime to dynamically make the decision. This is useful for tests that may require costly imports, to delay the cost until the test suite is actually executed.

**Parameters**

**skip_condition**

[bool or callable] Flag to determine whether to skip the decorated test.

**msg**

[str, optional] Message to give on raising a SkipTest exception. Default is None.

**Returns**

**decorator**

[function] Decorator which, when applied to a function, causes SkipTest to be raised when *skip_condition* is True, and the function to be called normally otherwise.

## Notes

The decorator itself is decorated with the `nose.tools.make_decorator` function in order to transmit function name, and various other metadata.

`numpy.testing.dec.`**`slow`**`(`*t*`)`

Label a test as 'slow'.

The exact definition of a slow test is obviously both subjective and hardware-dependent, but in general any individual test that requires more than a second or two should be labeled as slow (the whole suite consists of thousands of tests, so even a second is significant).

**Parameters**

**t**

[callable] The test to label as slow.

**Returns**

**t**

[callable] The decorated test *t*.

## Examples

The *`numpy.testing`* module includes `import decorators as dec`. A test can be decorated as slow like this:

```
from numpy.testing import *

@dec.slow
def test_big(self):
    print('Big, slow test')
```

`numpy.testing.`**`decorate_methods`**`(`*cls*, *decorator*, *testmatch=None*`)`

Apply a decorator to all methods in a class matching a regular expression.

The given decorator is applied to all public methods of *cls* that are matched by the regular expression *testmatch* (`testmatch.search(methodname)`). Methods that are private, i.e. start with an underscore, are ignored.

**Parameters**

**cls**

[class] Class whose methods to decorate.

**decorator**

[function] Decorator to apply to methods

**testmatch**

[compiled regexp or str, optional] The regular expression. Default value is None, in which case the nose default (`re.compile(r'(?:^|[\b_\.%s-])[Tt]est' % os.sep)`) is used. If *testmatch* is a string, it is compiled to a regular expression first.

## 4.28.3 Test Running

| | |
|---|---|
| *Tester* | alias of `numpy.testing._private.nosetester.NoseTester` |
| *run_module_suite*([file_to_run, argv]) | Run a test module. |
| *rundocs*([filename, raise_on_error]) | Run doctests found in the given file. |
| *suppress_warnings*([forwarding_rule]) | Context manager and decorator doing much the same as `warnings.catch_warnings`. |

`numpy.testing.`**`Tester`**

alias of `numpy.testing._private.nosetester.NoseTester`

`numpy.testing.`**`run_module_suite`**(*file_to_run=None*, *argv=None*)

Run a test module.

Equivalent to calling `$ nosetests <argv> <file_to_run>` from the command line

**Parameters**

**file_to_run**

[str, optional] Path to test module, or None. By default, run the module from which this function is called.

**argv**

[list of strings] Arguments to be passed to the nose test runner. `argv[0]` is ignored. All command line arguments accepted by `nosetests` will work. If it is the default value None, sys.argv is used.

New in version 1.9.0.

### Examples

Adding the following:

```
if __name__ == "__main__" :
    run_module_suite(argv=sys.argv)
```

at the end of a test module will run the tests when that module is called in the python interpreter.

Alternatively, calling:

```
>>> run_module_suite(file_to_run="numpy/tests/test_matlib.py")
```

from an interpreter will run all the test routine in 'test_matlib.py'.

`numpy.testing.`**`rundocs`**(*filename=None*, *raise_on_error=True*)

Run doctests found in the given file.

By default *rundocs* raises an AssertionError on failure.

**Parameters**

**filename**

[str] The path to the file for which the doctests are run.

**raise_on_error**

[bool] Whether to raise an AssertionError when a doctest fails. Default is True.

### Notes

The doctests can be run by the user/developer by adding the `doctests` argument to the `test()` call. For example, to run all tests (including doctests) for `numpy.lib`:

```
>>> np.lib.test(doctests=True)
```

**class** `numpy.testing.`**`suppress_warnings`**(*forwarding_rule='always'*)

Context manager and decorator doing much the same as `warnings.catch_warnings`.

However, it also provides a filter mechanism to work around https://bugs.python.org/issue4180.

This bug causes Python before 3.4 to not reliably show warnings again after they have been ignored once (even within catch_warnings). It means that no "ignore" filter can be used easily, since following tests might need to see the warning. Additionally it allows easier specificity for testing warnings and can be nested.

**Parameters**

**forwarding_rule**

[str, optional] One of "always", "once", "module", or "location". Analogous to the usual warnings module filter mode, it is useful to reduce noise mostly on the outmost level. Unsuppressed and unrecorded warnings will be forwarded based on this rule. Defaults to "always". "location" is equivalent to the warnings "default", match by exact location the warning warning originated from.

### Notes

Filters added inside the context manager will be discarded again when leaving it. Upon entering all filters defined outside a context will be applied automatically.

When a recording filter is added, matching warnings are stored in the `log` attribute as well as in the list returned by `record`.

If filters are added and the `module` keyword is given, the warning registry of this module will additionally be cleared when applying it, entering the context, or exiting it. This could cause warnings to appear a second time after leaving the context if they were configured to be printed once (default) and were already printed before the context was entered.

Nesting this context manager will work as expected when the forwarding rule is "always" (default). Unfiltered and unrecorded warnings will be passed out and be matched by the outer level. On the outmost level they will be printed (or caught by another warnings context). The forwarding rule argument can modify this behaviour.

Like `catch_warnings` this context manager is not threadsafe.

### Examples

With a context manager:

```
with np.testing.suppress_warnings() as sup:
    sup.filter(DeprecationWarning, "Some text")
    sup.filter(module=np.ma.core)
    log = sup.record(FutureWarning, "Does this occur?")
    command_giving_warnings()
    # The FutureWarning was given once, the filtered warnings were
    # ignored. All other warnings abide outside settings (may be
    # printed/error)
    assert_(len(log) == 1)
    assert_(len(sup.log) == 1)  # also stored in log attribute
```

Or as a decorator:

```
sup = np.testing.suppress_warnings()
sup.filter(module=np.ma.core)  # module must match exactly
@sup
def some_function():
    # do something which causes a warning in np.ma.core
    pass
```

### Methods

| | |
|---|---|
| `__call__`(self, func) | Function decorator to apply certain suppressions to a whole function. |
| `filter`(self[, category, message, module]) | Add a new suppressing filter or apply it if the state is entered. |
| `record`(self[, category, message, module]) | Append a new recording filter or apply it if the state is entered. |

method

`suppress_warnings.`**`__call__`**(*self*, *func*)

Function decorator to apply certain suppressions to a whole function.

method

`suppress_warnings.`**`filter`**(*self*, *category=<class 'Warning'>*, *message="*, *module=None*)

Add a new suppressing filter or apply it if the state is entered.

**Parameters**

**category**

[class, optional] Warning class to filter

**message**

[string, optional] Regular expression matching the warning message.

**module**

[module, optional] Module to filter for. Note that the module (and its file) must match exactly and cannot be a submodule. This may make it unreliable for external modules.

### Notes

When added within a context, filters are only added inside the context and will be forgotten when the context is exited.

method

`suppress_warnings.`**`record`**(*self*, *category=<class 'Warning'>*, *message="*, *module=None*)

Append a new recording filter or apply it if the state is entered.

All warnings matching will be appended to the `log` attribute.

**Parameters**

**category**

[class, optional] Warning class to filter

**message**

[string, optional] Regular expression matching the warning message.

**module**

[module, optional] Module to filter for. Note that the module (and its file) must match exactly and cannot be a submodule. This may make it unreliable for external modules.

**Returns**

**log**

[list] A list which will be filled with all matched warnings.

### Notes

When added within a context, filters are only added inside the context and will be forgotten when the context is exited.

## 4.28.4 Guidelines

### Testing Guidelines

### Introduction

Until the 1.15 release, NumPy used the nose testing framework, it now uses the pytest framework. The older framework is still maintained in order to support downstream projects that use the old numpy framework, but all tests for NumPy should use pytest.

Our goal is that every module and package in SciPy and NumPy should have a thorough set of unit tests. These tests should exercise the full functionality of a given routine as well as its robustness to erroneous or unexpected input arguments. Long experience has shown that by far the best time to write the tests is before you write or change the code - this is test-driven development. The arguments for this can sound rather abstract, but we can assure you that you will find that writing the tests first leads to more robust and better designed code. Well-designed tests with good coverage make an enormous difference to the ease of refactoring. Whenever a new bug is found in a routine, you should write a new test for that specific case and add it to the test suite to prevent that bug from creeping back in unnoticed.

To run SciPy's full test suite, use the following:

```
>>> import scipy
>>> scipy.test()
```

or from the command line:

```
$ python runtests.py
```

SciPy uses the testing framework from *numpy.testing*, so all the SciPy examples shown here are also applicable to NumPy. NumPy's full test suite can be run as follows:

```
>>> import numpy
>>> numpy.test()
```

The test method may take two or more arguments; the first, `label` is a string specifying what should be tested and the second, `verbose` is an integer giving the level of output verbosity. See the docstring for numpy.test for details. The default value for `label` is 'fast' - which will run the standard tests. The string 'full' will run the full battery of tests, including those identified as being slow to run. If `verbose` is 1 or less, the tests will just show information messages about the tests that are run; but if it is greater than 1, then the tests will also provide warnings on missing tests. So if you want to run every test and get messages about which modules don't have tests:

```
>>> scipy.test(label='full', verbose=2) # or scipy.test('full', 2)
```

Finally, if you are only interested in testing a subset of SciPy, for example, the `integrate` module, use the following:

```
>>> scipy.integrate.test()
```

or from the command line:

```
$python runtests.py -t scipy/integrate/tests
```

The rest of this page will give you a basic idea of how to add unit tests to modules in SciPy. It is extremely important for us to have extensive unit testing since this code is going to be used by scientists and researchers and is being developed by a large number of people spread across the world. So, if you are writing a package that you'd like to become part of SciPy, please write the tests as you develop the package. Also since much of SciPy is legacy code that was originally written without unit tests, there are still several modules that don't have tests yet. Please feel free to choose one of these modules and develop tests for it as you read through this introduction.

### Writing your own tests

Every Python module, extension module, or subpackage in the SciPy package directory should have a corresponding `test_<name>.py` file. Pytest examines these files for test methods (named test*) and test classes (named Test*).

Suppose you have a SciPy module `scipy/xxx/yyy.py` containing a function `zzz()`. To test this function you would create a test module called `test_yyy.py`. If you only need to test one aspect of `zzz`, you can simply add a test function:

```
def test_zzz():
    assert_(zzz() == 'Hello from zzz')
```

More often, we need to group a number of tests together, so we create a test class:

```
from numpy.testing import assert_, assert_raises

# import xxx symbols
from scipy.xxx.yyy import zzz

class TestZzz:
```

(continues on next page)

(continued from previous page)

```
    def test_simple(self):
        assert_(zzz() == 'Hello from zzz')

    def test_invalid_parameter(self):
        assert_raises(...)
```

Within these test methods, `assert_()` and related functions are used to test whether a certain assumption is valid. If the assertion fails, the test fails. Note that the Python builtin `assert` should not be used, because it is stripped during compilation with `-O`.

Note that `test_` functions or methods should not have a docstring, because that makes it hard to identify the test from the output of running the test suite with `verbose=2` (or similar verbosity setting). Use plain comments (`#`) if necessary.

### Labeling tests

As an alternative to `pytest.mark.<label>`, there are a number of labels you can use.

Unlabeled tests like the ones above are run in the default `scipy.test()` run. If you want to label your test as slow - and therefore reserved for a full `scipy.test(label='full')` run, you can label it with a decorator:

```
# numpy.testing module includes 'import decorators as dec'
from numpy.testing import dec, assert_

@dec.slow
def test_big(self):
    print 'Big, slow test'
```

Similarly for methods:

```
class test_zzz:
    @dec.slow
    def test_simple(self):
        assert_(zzz() == 'Hello from zzz')
```

Available labels are:

- `slow`: marks a test as taking a long time
- `setastest(tf)`: work-around for test discovery when the test name is non conformant
- `skipif(condition, msg=None)`: skips the test when `eval(condition)` is `True`
- `knownfailureif(fail_cond, msg=None)`: will avoid running the test if `eval(fail_cond)` is `True`, useful for tests that conditionally segfault
- `deprecated(conditional=True)`: filters deprecation warnings emitted in the test
- `paramaterize(var, input)`: an alternative to pytest.mark.paramaterized

### Easier setup and teardown functions / methods

Testing looks for module-level or class-level setup and teardown functions by name; thus:

```
def setup():
    """Module-level setup"""
    print 'doing setup'

def teardown():
    """Module-level teardown"""
    print 'doing teardown'


class TestMe:
    def setup():
        """Class-level setup"""
        print 'doing setup'

    def teardown():
        """Class-level teardown"""
        print 'doing teardown'
```

Setup and teardown functions to functions and methods are known as "fixtures", and their use is not encouraged.

### Parametric tests

One very nice feature of testing is allowing easy testing across a range of parameters - a nasty problem for standard unit tests. Use the `dec.paramaterize` decorator.

### Doctests

Doctests are a convenient way of documenting the behavior of a function and allowing that behavior to be tested at the same time. The output of an interactive Python session can be included in the docstring of a function, and the test framework can run the example and compare the actual output to the expected output.

The doctests can be run by adding the `doctests` argument to the `test()` call; for example, to run all tests (including doctests) for numpy.lib:

```
>>> import numpy as np
>>> np.lib.test(doctests=True)
```

The doctests are run as if they are in a fresh Python instance which has executed `import numpy as np`. Tests that are part of a SciPy subpackage will have that subpackage already imported. E.g. for a test in `scipy/linalg/tests/`, the namespace will be created such that `from scipy import linalg` has already executed.

### `tests/`

Rather than keeping the code and the tests in the same directory, we put all the tests for a given subpackage in a `tests/` subdirectory. For our example, if it doesn't already exist you will need to create a `tests/` directory in `scipy/xxx/`. So the path for `test_yyy.py` is `scipy/xxx/tests/test_yyy.py`.

Once the `scipy/xxx/tests/test_yyy.py` is written, its possible to run the tests by going to the `tests/` directory and typing:

```
python test_yyy.py
```

Or if you add `scipy/xxx/tests/` to the Python path, you could run the tests interactively in the interpreter like this:

```
>>> import test_yyy
>>> test_yyy.test()
```

### `__init__.py` and `setup.py`

Usually, however, adding the `tests/` directory to the python path isn't desirable. Instead it would better to invoke the test straight from the module `xxx`. To this end, simply place the following lines at the end of your package's `__init__.py` file:

```
...
def test(level=1, verbosity=1):
    from numpy.testing import Tester
    return Tester().test(level, verbosity)
```

You will also need to add the tests directory in the configuration section of your setup.py:

```
...
def configuration(parent_package='', top_path=None):
    ...
    config.add_subpackage('tests')
    return config
...
```

Now you can do the following to test your module:

```
>>> import scipy
>>> scipy.xxx.test()
```

Also, when invoking the entire SciPy test suite, your tests will be found and run:

```
>>> import scipy
>>> scipy.test()
# your tests are included and run automatically!
```

## Tips & Tricks

### Creating many similar tests

If you have a collection of tests that must be run multiple times with minor variations, it can be helpful to create a base class containing all the common tests, and then create a subclass for each variation. Several examples of this technique exist in NumPy; below are excerpts from one in numpy/linalg/tests/test_linalg.py:

```
class LinalgTestCase:
    def test_single(self):
        a = array([[1.,2.], [3.,4.]], dtype=single)
        b = array([2., 1.], dtype=single)
        self.do(a, b)

    def test_double(self):
        a = array([[1.,2.], [3.,4.]], dtype=double)
        b = array([2., 1.], dtype=double)
        self.do(a, b)

    ...

class TestSolve(LinalgTestCase):
    def do(self, a, b):
        x = linalg.solve(a, b)
        assert_almost_equal(b, dot(a, x))
        assert_(imply(isinstance(b, matrix), isinstance(x, matrix)))

class TestInv(LinalgTestCase):
    def do(self, a, b):
        a_inv = linalg.inv(a)
        assert_almost_equal(dot(a, a_inv), identity(asarray(a).shape[0]))
        assert_(imply(isinstance(a, matrix), isinstance(a_inv, matrix)))
```

In this case, we wanted to test solving a linear algebra problem using matrices of several data types, using `linalg.solve` and `linalg.inv`. The common test cases (for single-precision, double-precision, etc. matrices) are collected in `LinalgTestCase`.

### Known failures & skipping tests

Sometimes you might want to skip a test or mark it as a known failure, such as when the test suite is being written before the code it's meant to test, or if a test only fails on a particular architecture.

To skip a test, simply use `skipif`:

```
import pytest

@pytest.mark.skipif(SkipMyTest, reason="Skipping this test because...")
def test_something(foo):
    ...
```

The test is marked as skipped if `SkipMyTest` evaluates to nonzero, and the message in verbose test output is the second argument given to `skipif`. Similarly, a test can be marked as a known failure by using `xfail`:

```
import pytest

@pytest.mark.xfail(MyTestFails, reason="This test is known to fail because...")
```

(continues on next page)

(continued from previous page)

```
def test_something_else(foo):
    ...
```

Of course, a test can be unconditionally skipped or marked as a known failure by using `skip` or `xfail` without argument, respectively.

A total of the number of skipped and known failing tests is displayed at the end of the test run. Skipped tests are marked as `'S'` in the test results (or `'SKIPPED'` for `verbose > 1`), and known failing tests are marked as `'x'` (or `'XFAIL'` if `verbose > 1`).

### Tests on random data

Tests on random data are good, but since test failures are meant to expose new bugs or regressions, a test that passes most of the time but fails occasionally with no code changes is not helpful. Make the random data deterministic by setting the random number seed before generating it. Use either Python's `random.seed(some_number)` or NumPy's `numpy.random.seed(some_number)`, depending on the source of random numbers.

Alternatively, you can use Hypothesis to generate arbitrary data. Hypothesis manages both Python's and Numpy's random seeds for you, and provides a very concise and powerful way to describe data (including `hypothesis.extra.numpy`, e.g. for a set of mutually-broadcastable shapes).

The advantages over random generation include tools to replay and share failures without requiring a fixed seed, reporting *minimal* examples for each failure, and better-than-naive-random techniques for triggering bugs.

# 4.29 Window functions

## 4.29.1 Various windows

| | |
|---|---|
| `bartlett`(M) | Return the Bartlett window. |
| `blackman`(M) | Return the Blackman window. |
| `hamming`(M) | Return the Hamming window. |
| `hanning`(M) | Return the Hanning window. |
| `kaiser`(M, beta) | Return the Kaiser window. |

`numpy.`**`bartlett`**`(`*M*`)`

Return the Bartlett window.

The Bartlett window is very similar to a triangular window, except that the end points are at zero. It is often used in signal processing for tapering a signal, without generating too much ripple in the frequency domain.

**Parameters**

**M**

[int] Number of points in the output window. If zero or less, an empty array is returned.

**Returns**

**out**

[array] The triangular window, with the maximum value normalized to one (the value one appears only if the number of samples is odd), with the first and last samples equal to zero.

**See also:**

*blackman*, *hamming*, *hanning*, *kaiser*

## Notes

The Bartlett window is defined as

$$w(n) = \frac{2}{M-1}\left(\frac{M-1}{2} - \left|n - \frac{M-1}{2}\right|\right)$$

Most references to the Bartlett window come from the signal processing literature, where it is used as one of many windowing functions for smoothing values. Note that convolution with this window produces linear interpolation. It is also known as an apodization (which means"removing the foot", i.e. smoothing discontinuities at the beginning and end of the sampled signal) or tapering function. The fourier transform of the Bartlett is the product of two sinc functions. Note the excellent discussion in Kanasewich.

## References

[1], [2], [3], [4], [5]

## Examples

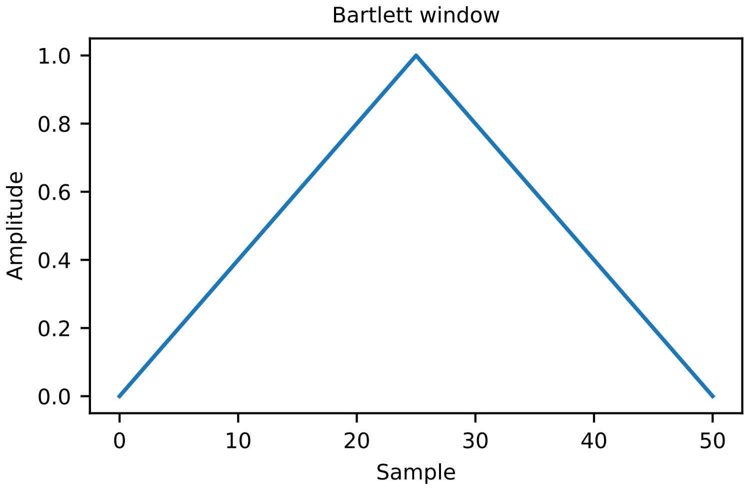
```
>>> import matplotlib.pyplot as plt
>>> np.bartlett(12)
array([ 0.        ,  0.18181818,  0.36363636,  0.54545455,  0.72727273, # may vary
        0.90909091,  0.90909091,  0.72727273,  0.54545455,  0.36363636,
        0.18181818,  0.        ])
```

Plot the window and its frequency response (requires SciPy and matplotlib):

```
>>> from numpy.fft import fft, fftshift
>>> window = np.bartlett(51)
>>> plt.plot(window)
[<matplotlib.lines.Line2D object at 0x...>]
>>> plt.title("Bartlett window")
Text(0.5, 1.0, 'Bartlett window')
>>> plt.ylabel("Amplitude")
Text(0, 0.5, 'Amplitude')
>>> plt.xlabel("Sample")
Text(0.5, 0, 'Sample')
>>> plt.show()
```

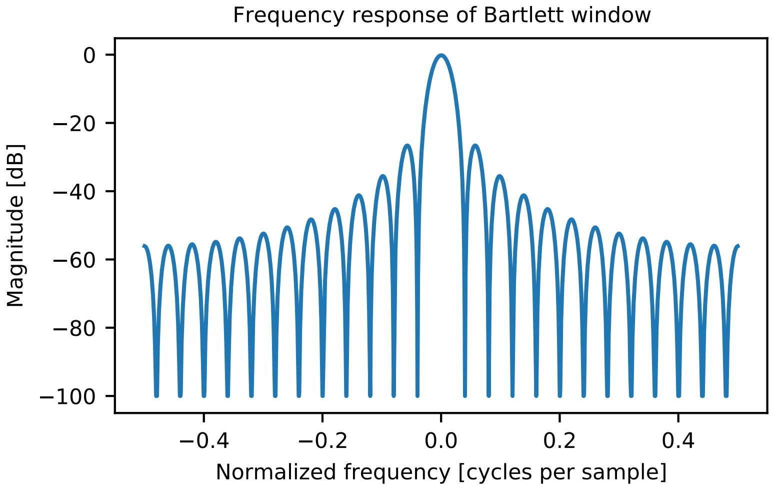
```
>>> plt.figure()
<Figure size 640x480 with 0 Axes>
>>> A = fft(window, 2048) / 25.5
>>> mag = np.abs(fftshift(A))
>>> freq = np.linspace(-0.5, 0.5, len(A))
>>> with np.errstate(divide='ignore', invalid='ignore'):
...     response = 20 * np.log10(mag)
...
>>> response = np.clip(response, -100, 100)
>>> plt.plot(freq, response)
[<matplotlib.lines.Line2D object at 0x...>]
```

(continues on next page)



(continued from previous page)

```
>>> plt.title("Frequency response of Bartlett window")
Text(0.5, 1.0, 'Frequency response of Bartlett window')
>>> plt.ylabel("Magnitude [dB]")
Text(0, 0.5, 'Magnitude [dB]')
>>> plt.xlabel("Normalized frequency [cycles per sample]")
Text(0.5, 0, 'Normalized frequency [cycles per sample]')
>>> _ = plt.axis('tight')
>>> plt.show()
```



`numpy.`**`blackman`**`(`*M*`)`

Return the Blackman window.

The Blackman window is a taper formed by using the first three terms of a summation of cosines. It was designed to have close to the minimal leakage possible. It is close to optimal, only slightly worse than a Kaiser window.

**Parameters**

**M**

[int] Number of points in the output window. If zero or less, an empty array is returned.

**Returns**

**out**

[ndarray] The window, with the maximum value normalized to one (the value one appears only if the number of samples is odd).

**See also:**

*bartlett*, *hamming*, *hanning*, *kaiser*

### Notes

The Blackman window is defined as

$$w(n) = 0.42 - 0.5\cos(2\pi n/M) + 0.08\cos(4\pi n/M)$$

Most references to the Blackman window come from the signal processing literature, where it is used as one of many windowing functions for smoothing values. It is also known as an apodization (which means "removing the foot", i.e. smoothing discontinuities at the beginning and end of the sampled signal) or tapering function. It is known as a "near optimal" tapering function, almost as good (by some measures) as the kaiser window.

### References

Blackman, R.B. and Tukey, J.W., (1958) The measurement of power spectra, Dover Publications, New York.

Oppenheim, A.V., and R.W. Schafer. Discrete-Time Signal Processing. Upper Saddle River, NJ: Prentice-Hall, 1999, pp. 468-471.

### Examples

```
>>> import matplotlib.pyplot as plt
>>> np.blackman(12)
array([-1.38777878e-17,   3.26064346e-02,   1.59903635e-01, # may vary
        4.14397981e-01,   7.36045180e-01,   9.67046769e-01,
        9.67046769e-01,   7.36045180e-01,   4.14397981e-01,
        1.59903635e-01,   3.26064346e-02,  -1.38777878e-17])
```

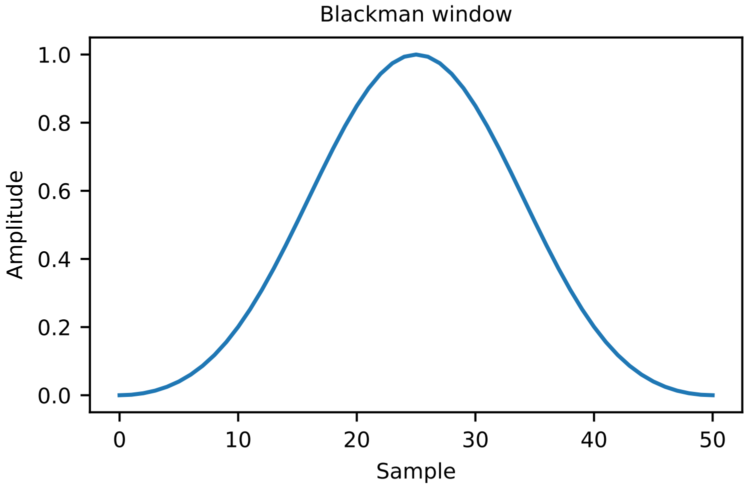
Plot the window and the frequency response:

```
>>> from numpy.fft import fft, fftshift
>>> window = np.blackman(51)
>>> plt.plot(window)
[<matplotlib.lines.Line2D object at 0x...>]
>>> plt.title("Blackman window")
Text(0.5, 1.0, 'Blackman window')
>>> plt.ylabel("Amplitude")
Text(0, 0.5, 'Amplitude')
```

(continues on next page)

(continued from previous page)

```
>>> plt.xlabel("Sample")
Text(0.5, 0, 'Sample')
>>> plt.show()
```



```
>>> plt.figure()
<Figure size 640x480 with 0 Axes>
>>> A = fft(window, 2048) / 25.5
>>> mag = np.abs(fftshift(A))
>>> freq = np.linspace(-0.5, 0.5, len(A))
>>> with np.errstate(divide='ignore', invalid='ignore'):
...     response = 20 * np.log10(mag)
...
>>> response = np.clip(response, -100, 100)
>>> plt.plot(freq, response)
[<matplotlib.lines.Line2D object at 0x...>]
>>> plt.title("Frequency response of Blackman window")
Text(0.5, 1.0, 'Frequency response of Blackman window')
>>> plt.ylabel("Magnitude [dB]")
Text(0, 0.5, 'Magnitude [dB]')
>>> plt.xlabel("Normalized frequency [cycles per sample]")
Text(0.5, 0, 'Normalized frequency [cycles per sample]')
>>> _ = plt.axis('tight')
>>> plt.show()
```

`numpy.`**`hamming`**`(`*M*`)`

Return the Hamming window.

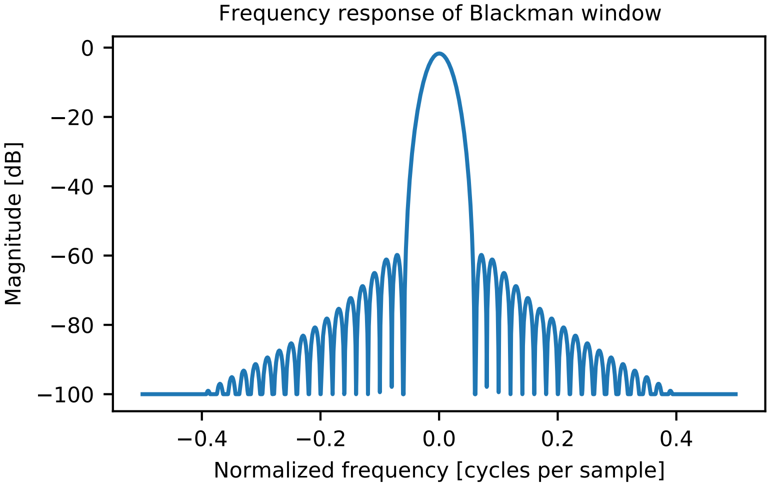
The Hamming window is a taper formed by using a weighted cosine.

**Parameters**

**M**

[int] Number of points in the output window. If zero or less, an empty array is returned.

**Returns**



**out**

[ndarray] The window, with the maximum value normalized to one (the value one appears only if the number of samples is odd).

**See also:**

`bartlett`, `blackman`, `hanning`, `kaiser`

## Notes

The Hamming window is defined as

$$w(n) = 0.54 - 0.46cos\left(\frac{2\pi n}{M-1}\right) \qquad 0 \leq n \leq M-1$$

The Hamming was named for R. W. Hamming, an associate of J. W. Tukey and is described in Blackman and Tukey. It was recommended for smoothing the truncated autocovariance function in the time domain. Most references to the Hamming window come from the signal processing literature, where it is used as one of many windowing functions for smoothing values. It is also known as an apodization (which means "removing the foot", i.e. smoothing discontinuities at the beginning and end of the sampled signal) or tapering function.

## References

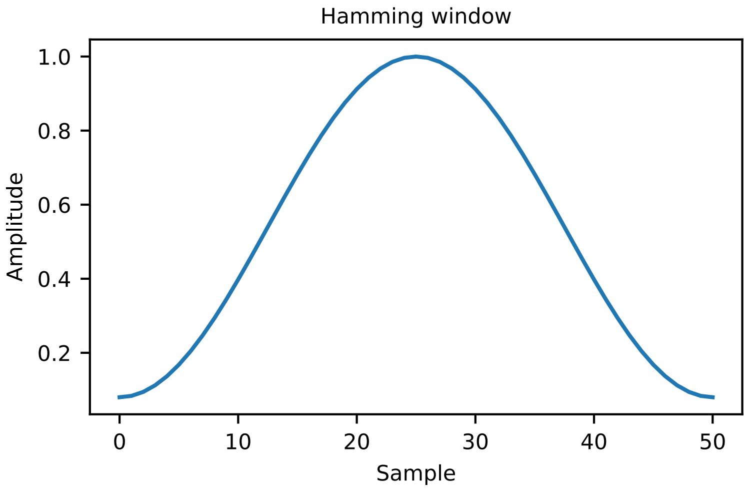
[1], [2], [3], [4]

## Examples

```
>>> np.hamming(12)
array([ 0.08      ,  0.15302337,  0.34890909,  0.60546483,  0.84123594, # may vary
        0.98136677,  0.98136677,  0.84123594,  0.60546483,  0.34890909,
        0.15302337,  0.08      ])
```

Plot the window and the frequency response:

```
>>> import matplotlib.pyplot as plt
>>> from numpy.fft import fft, fftshift
>>> window = np.hamming(51)
>>> plt.plot(window)
[<matplotlib.lines.Line2D object at 0x...>]
>>> plt.title("Hamming window")
Text(0.5, 1.0, 'Hamming window')
>>> plt.ylabel("Amplitude")
Text(0, 0.5, 'Amplitude')
>>> plt.xlabel("Sample")
Text(0.5, 0, 'Sample')
>>> plt.show()
```

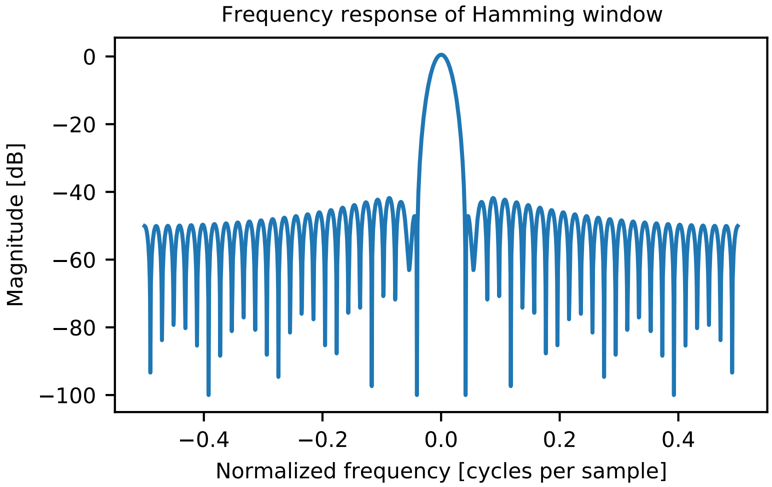


```
>>> plt.figure()
<Figure size 640x480 with 0 Axes>
>>> A = fft(window, 2048) / 25.5
>>> mag = np.abs(fftshift(A))
>>> freq = np.linspace(-0.5, 0.5, len(A))
>>> response = 20 * np.log10(mag)
>>> response = np.clip(response, -100, 100)
>>> plt.plot(freq, response)
[<matplotlib.lines.Line2D object at 0x...>]
>>> plt.title("Frequency response of Hamming window")
Text(0.5, 1.0, 'Frequency response of Hamming window')
>>> plt.ylabel("Magnitude [dB]")
Text(0, 0.5, 'Magnitude [dB]')
>>> plt.xlabel("Normalized frequency [cycles per sample]")
```

(continues on next page)

(continued from previous page)

```
Text(0.5, 0, 'Normalized frequency [cycles per sample]')
>>> plt.axis('tight')
...
>>> plt.show()
```



`numpy.`**`hanning`**`(`*M*`)`

Return the Hanning window.

The Hanning window is a taper formed by using a weighted cosine.

**Parameters**

**M**

[int] Number of points in the output window. If zero or less, an empty array is returned.

**Returns**

**out**

[ndarray, shape(M,)] The window, with the maximum value normalized to one (the value one appears only if *M* is odd).

**See also:**

*`bartlett`*, *`blackman`*, *`hamming`*, *`kaiser`*

## Notes

The Hanning window is defined as

$$w(n) = 0.5 - 0.5cos\left(\frac{2\pi n}{M-1}\right) \qquad 0 \leq n \leq M-1$$

The Hanning was named for Julius von Hann, an Austrian meteorologist. It is also known as the Cosine Bell. Some authors prefer that it be called a Hann window, to help avoid confusion with the very similar Hamming window.

Most references to the Hanning window come from the signal processing literature, where it is used as one of many windowing functions for smoothing values. It is also known as an apodization (which means "removing the foot", i.e. smoothing discontinuities at the beginning and end of the sampled signal) or tapering function.

## References

[1], [2], [3], [4]

## Examples

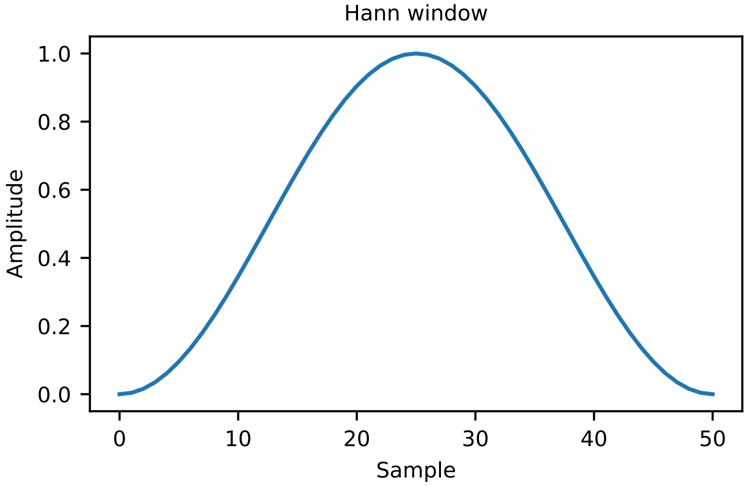
```
>>> np.hanning(12)
array([0.        , 0.07937323, 0.29229249, 0.57115742, 0.82743037,
       0.97974649, 0.97974649, 0.82743037, 0.57115742, 0.29229249,
       0.07937323, 0.        ])
```

Plot the window and its frequency response:

```
>>> import matplotlib.pyplot as plt
>>> from numpy.fft import fft, fftshift
>>> window = np.hanning(51)
>>> plt.plot(window)
[<matplotlib.lines.Line2D object at 0x...>]
>>> plt.title("Hann window")
Text(0.5, 1.0, 'Hann window')
>>> plt.ylabel("Amplitude")
Text(0, 0.5, 'Amplitude')
>>> plt.xlabel("Sample")
Text(0.5, 0, 'Sample')
>>> plt.show()
```

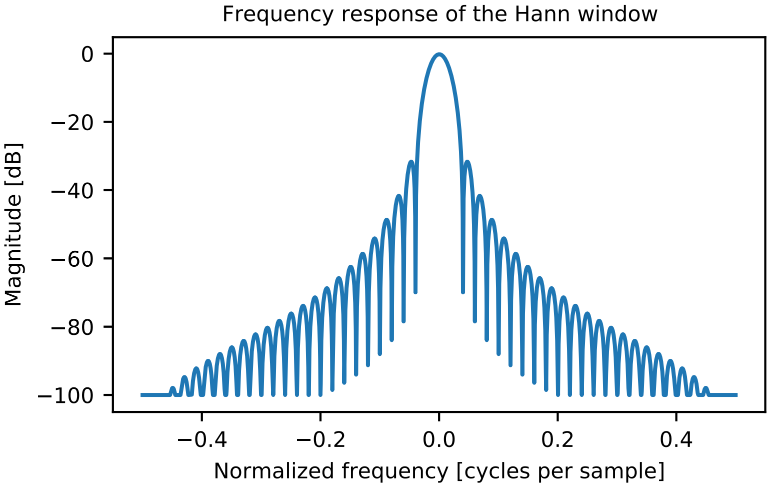
```
>>> plt.figure()
<Figure size 640x480 with 0 Axes>
>>> A = fft(window, 2048) / 25.5
>>> mag = np.abs(fftshift(A))
>>> freq = np.linspace(-0.5, 0.5, len(A))
>>> with np.errstate(divide='ignore', invalid='ignore'):
...     response = 20 * np.log10(mag)
...
>>> response = np.clip(response, -100, 100)
>>> plt.plot(freq, response)
[<matplotlib.lines.Line2D object at 0x...>]
>>> plt.title("Frequency response of the Hann window")
Text(0.5, 1.0, 'Frequency response of the Hann window')
>>> plt.ylabel("Magnitude [dB]")
Text(0, 0.5, 'Magnitude [dB]')
```

(continues on next page)



(continued from previous page)

```
>>> plt.xlabel("Normalized frequency [cycles per sample]")
Text(0.5, 0, 'Normalized frequency [cycles per sample]')
>>> plt.axis('tight')
...
>>> plt.show()
```



`numpy.`**`kaiser`**(*M*, *beta*)

Return the Kaiser window.

The Kaiser window is a taper formed by using a Bessel function.

**Parameters**

**M**

[int] Number of points in the output window. If zero or less, an empty array is returned.

**beta**

[float] Shape parameter for window.

**Returns**

**out**

[array] The window, with the maximum value normalized to one (the value one appears only if the number of samples is odd).

**See also:**

`bartlett`, `blackman`, `hamming`, `hanning`

## Notes

The Kaiser window is defined as

$$w(n) = I_0\left(\beta\sqrt{1 - \frac{4n^2}{(M-1)^2}}\right)/I_0(\beta)$$

with

$$-\frac{M-1}{2} \leq n \leq \frac{M-1}{2},$$

where $I_0$ is the modified zeroth-order Bessel function.

The Kaiser was named for Jim Kaiser, who discovered a simple approximation to the DPSS window based on Bessel functions. The Kaiser window is a very good approximation to the Digital Prolate Spheroidal Sequence, or Slepian window, which is the transform which maximizes the energy in the main lobe of the window relative to total energy.

The Kaiser can approximate many other windows by varying the beta parameter.

| beta | Window shape |
|---|---|
| 0 | Rectangular |
| 5 | Similar to a Hamming |
| 6 | Similar to a Hanning |
| 8.6 | Similar to a Blackman |

A beta value of 14 is probably a good starting point. Note that as beta gets large, the window narrows, and so the number of samples needs to be large enough to sample the increasingly narrow spike, otherwise NaNs will get returned.

Most references to the Kaiser window come from the signal processing literature, where it is used as one of many windowing functions for smoothing values. It is also known as an apodization (which means "removing the foot", i.e. smoothing discontinuities at the beginning and end of the sampled signal) or tapering function.

### References

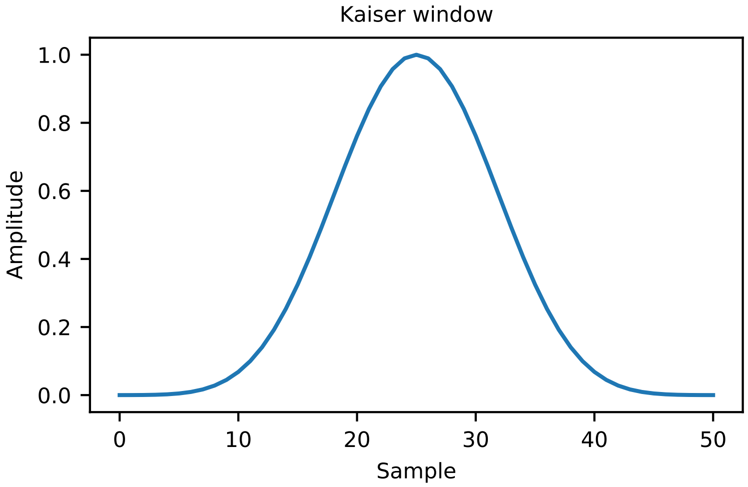
[1], [2], [3]

### Examples

```
>>> import matplotlib.pyplot as plt
>>> np.kaiser(12, 14)
 array([7.72686684e-06, 3.46009194e-03, 4.65200189e-02, # may vary
        2.29737120e-01, 5.99885316e-01, 9.45674898e-01,
        9.45674898e-01, 5.99885316e-01, 2.29737120e-01,
        4.65200189e-02, 3.46009194e-03, 7.72686684e-06])
```

Plot the window and the frequency response:

```
>>> from numpy.fft import fft, fftshift
>>> window = np.kaiser(51, 14)
>>> plt.plot(window)
[<matplotlib.lines.Line2D object at 0x...>]
>>> plt.title("Kaiser window")
Text(0.5, 1.0, 'Kaiser window')
>>> plt.ylabel("Amplitude")
Text(0, 0.5, 'Amplitude')
>>> plt.xlabel("Sample")
Text(0.5, 0, 'Sample')
>>> plt.show()
```

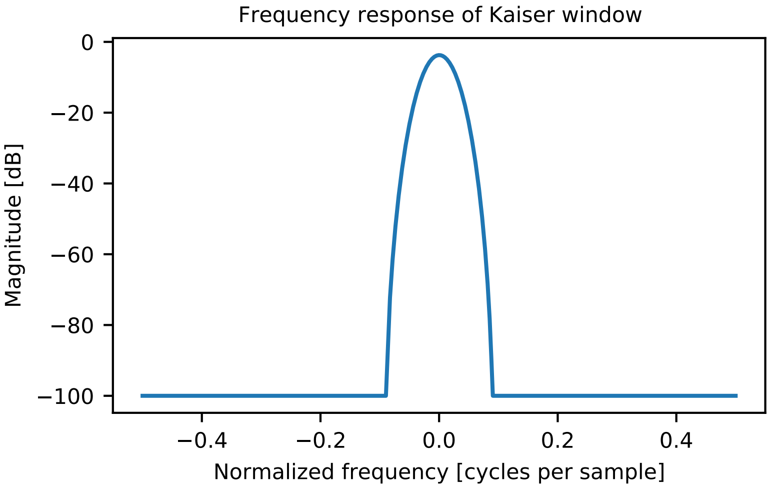


```
>>> plt.figure()
<Figure size 640x480 with 0 Axes>
>>> A = fft(window, 2048) / 25.5
>>> mag = np.abs(fftshift(A))
>>> freq = np.linspace(-0.5, 0.5, len(A))
>>> response = 20 * np.log10(mag)
>>> response = np.clip(response, -100, 100)
>>> plt.plot(freq, response)
```

(continues on next page)

(continued from previous page)

```
[<matplotlib.lines.Line2D object at 0x...>]
>>> plt.title("Frequency response of Kaiser window")
Text(0.5, 1.0, 'Frequency response of Kaiser window')
>>> plt.ylabel("Magnitude [dB]")
Text(0, 0.5, 'Magnitude [dB]')
>>> plt.xlabel("Normalized frequency [cycles per sample]")
Text(0.5, 0, 'Normalized frequency [cycles per sample]')
>>> plt.axis('tight')
(-0.5, 0.5, -100.0, ...) # may vary
>>> plt.show()
```



CHAPTER

FIVE

# GLOBAL STATE

NumPy has a few import-time, compile-time, or runtime options which change the global behaviour. Most of these are related to performance or for debugging purposes and will not be interesting to the vast majority of users.

## 5.1 Performance-Related Options

### 5.1.1 Number of Threads used for Linear Algebra

NumPy itself is normally intentionally limited to a single thread during function calls, however it does support multiple Python threads running at the same time. Note that for performant linear algebra NumPy uses a BLAS backend such as OpenBLAS or MKL, which may use multiple threads that may be controlled by environment variables such as `OMP_NUM_THREADS` depending on what is used. One way to control the number of threads is the package threadpoolctl

### 5.1.2 Madvise Hugepage on Linux

When working with very large arrays on modern Linux kernels, you can experience a significant speedup when transparent hugepage is used. The current system policy for transparent hugepages can be seen by:

```
cat /sys/kernel/mm/transparent_hugepage/enabled
```

When set to `madvise` NumPy will typically use hugepages for a performance boost. This behaviour can be modified by setting the environment variable:

```
NUMPY_MADVISE_HUGEPAGE=0
```

or setting it to `1` to always enable it. When not set, the default is to use madvise on Kernels 4.6 and newer. These kernels presumably experience a large speedup with hugepage support. This flag is checked at import time.

## 5.2 Interoperability-Related Options

The array function protocol which allows array-like objects to hook into the NumPy API is currently enabled by default. This option exists since NumPy 1.16 and is enabled by default since NumPy 1.17. It can be disabled using:

```
NUMPY_EXPERIMENTAL_ARRAY_FUNCTION=0
```

See also *`numpy.class.__array_function__`* for more information. This flag is checked at import time.

## 5.3 Debugging-Related Options

### 5.3.1 Relaxed Strides Checking

The *compile-time* environment variables:

```
NPY_RELAXED_STRIDES_DEBUG=0
NPY_RELAXED_STRIDES_CHECKING=1
```

control how NumPy reports contiguity for arrays. The default that it is enabled and the debug mode is disabled. This setting should always be enabled. Setting the debug option can be interesting for testing code written in C which iterates through arrays that may or may not be contiguous in memory. Most users will have no reason to change these, for details please see the memory layout documentation.

CHAPTER

SIX

# PACKAGING (`NUMPY.DISTUTILS`)

NumPy provides enhanced distutils functionality to make it easier to build and install sub-packages, auto-generate code, and extension modules that use Fortran-compiled libraries. To use features of NumPy distutils, use the `setup` command from `numpy.distutils.core`. A useful `Configuration` class is also provided in `numpy.distutils.misc_util` that can make it easier to construct keyword arguments to pass to the setup function (by passing the dictionary obtained from the todict() method of the class). More information is available in the *NumPy Distutils - Users Guide*.

The choice and location of linked libraries such as BLAS and LAPACK as well as include paths and other such build options can be specified in a `site.cfg` file located in the NumPy root repository or a `.numpy-site.cfg` file in your home directory. See the `site.cfg.example` example file included in the NumPy repository or sdist for documentation.

## 6.1 Modules in `numpy.distutils`

### 6.1.1 distutils.misc_util

`numpy.distutils.misc_util.`**`get_numpy_include_dirs`**`()`

`numpy.distutils.misc_util.`**`default_config_dict`**`(`*name=None*, *parent_name=None*, *local_path=None*`)`

Return a configuration dictionary for usage in configuration() function defined in file setup_<name>.py.

`numpy.distutils.misc_util.`**`dict_append`**`(`*d*, ***kws*`)`

`numpy.distutils.misc_util.`**`appendpath`**`(`*prefix*, *path*`)`

`numpy.distutils.misc_util.`**`generate_config_py`**`(`*target*`)`

Generate config.py file containing system_info information used during building the package.

**Usage:**

> config['py_modules'].append((packagename, '__config__',generate_config_py))

`numpy.distutils.misc_util.`**`get_cmd`**`(`*cmdname*, *_cache={}*`)`

`numpy.distutils.misc_util.`**`allpath`**`(`*name*`)`

Convert a /-separated pathname to one using the OS's path separator.

`numpy.distutils.misc_util.`**`get_mathlibs`**`(`*path=None*`)`

Return the MATHLIB line from numpyconfig.h

`numpy.distutils.misc_util.`**`terminal_has_colors`**`()`

`numpy.distutils.misc_util.`**`red_text`**`(`*s*`)`

`numpy.distutils.misc_util.`**`green_text`**`(`*s*`)` `numpy.distutils.misc_util.`**`yellow_text`**(*s*)

`numpy.distutils.misc_util.`**`blue_text`**(*s*)

`numpy.distutils.misc_util.`**`cyan_text`**(*s*)

`numpy.distutils.misc_util.`**`cyg2win32`**(*path*)

`numpy.distutils.misc_util.`**`mingw32`**()

Return true when using mingw32 environment.

`numpy.distutils.misc_util.`**`all_strings`**(*lst*)

Return True if all items in lst are string objects.

`numpy.distutils.misc_util.`**`has_f_sources`**(*sources*)

Return True if sources contains Fortran files

`numpy.distutils.misc_util.`**`has_cxx_sources`**(*sources*)

Return True if sources contains C++ files

`numpy.distutils.misc_util.`**`filter_sources`**(*sources*)

Return four lists of filenames containing C, C++, Fortran, and Fortran 90 module sources, respectively.

`numpy.distutils.misc_util.`**`get_dependencies`**(*sources*)

`numpy.distutils.misc_util.`**`is_local_src_dir`**(*directory*)

Return true if directory is local directory.

`numpy.distutils.misc_util.`**`get_ext_source_files`**(*ext*)

`numpy.distutils.misc_util.`**`get_script_files`**(*scripts*)

`numpy.distutils.misc_util.`**`get_lib_source_files`**(*lib*)

`numpy.distutils.misc_util.`**`get_data_files`**(*data*)

`numpy.distutils.misc_util.`**`dot_join`**(**args*)

`numpy.distutils.misc_util.`**`get_frame`**(*level=0*)

Return frame object from call stack with given level.

`numpy.distutils.misc_util.`**`minrelpath`**(*path*)

Resolve  and '.' from path.

`numpy.distutils.misc_util.`**`njoin`**(**path*)

Join two or more pathname components + - convert a /-separated pathname to one using the OS's path separator. - resolve  and  from path.

Either passing n arguments as in njoin('a','b'), or a sequence of n names as in njoin(['a','b']) is handled, or a mixture of such arguments.

`numpy.distutils.misc_util.`**`is_sequence`**(*seq*)

`numpy.distutils.misc_util.`**`is_string`**(*s*)

`numpy.distutils.misc_util.`**`as_list`**(*seq*)

`numpy.distutils.misc_util.`**`gpaths`**(*paths*, *local_path=”*, *include_non_existing=True*)

Apply glob to paths and prepend local_path if needed.

`numpy.distutils.misc_util.`**`get_language`**(*sources*)

Determine language value (c,f77,f90) from sources

`numpy.distutils.misc_util.`**`quote_args`**(*args*)

`numpy.distutils.misc_util.`**`get_build_architecture`**()

`numpy.distutils.misc_util.`**`get_info`**`(`*pkgname*, *dirs=None*`)`
Return an info dict for a given C library.

The info dict contains the necessary options to use the C library.

**Parameters**

**pkgname**

[str] Name of the package (should match the name of the .ini file, without the extension, e.g. foo for the file foo.ini).

**dirs**

[sequence, optional] If given, should be a sequence of additional directories where to look for npy-pkg-config files. Those directories are searched prior to the NumPy directory.

**Returns**

**info**

[dict] The dictionary with build information.

**Raises**

**PkgNotFound**

If the package is not found.

**See also:**

*`Configuration.add_npy_pkg_config`*, *`Configuration.add_installed_library`*, *`get_pkg_info`*

### Examples

To get the necessary information for the npymath library from NumPy:

```
>>> npymath_info = np.distutils.misc_util.get_info('npymath')
>>> npymath_info
{'define_macros': [], 'libraries': ['npymath'], 'library_dirs':
['.../numpy/core/lib'], 'include_dirs': ['.../numpy/core/include']}
```

This info dict can then be used as input to a *`Configuration`* instance:

```
config.add_extension('foo', sources=['foo.c'], extra_info=npymath_info)
```

`numpy.distutils.misc_util.`**`get_pkg_info`**`(`*pkgname*, *dirs=None*`)`
Return library info for the given package.

**Parameters**

**pkgname**

[str] Name of the package (should match the name of the .ini file, without the extension, e.g. foo for the file foo.ini).

**dirs**

[sequence, optional] If given, should be a sequence of additional directories where to look for npy-pkg-config files. Those directories are searched prior to the NumPy directory.

**Returns**

**pkginfo**

[class instance] The *LibraryInfo* instance containing the build information.

**Raises**

**PkgNotFound**

If the package is not found.

**See also:**

*`Configuration.add_npy_pkg_config`*, *`Configuration.add_installed_library`*, *`get_info`*

`numpy.distutils.misc_util.`**`get_num_build_jobs`**`()`

Get number of parallel build jobs set by the –parallel command line argument of setup.py If the command did not receive a setting the environment variable NPY_NUM_BUILD_JOBS is checked. If that is unset, return the number of processors on the system, with a maximum of 8 (to prevent overloading the system if there a lot of CPUs).

**Returns**

**out**

[int] number of parallel jobs that can be run

| | |
|---|---|
| *`ccompiler`* | |
| *`cpuinfo.cpu`* | |
| *`core.Extension`*(name, sources[, …]) | **Parameters** |
| *`exec_command`* | exec_command |
| *`log.set_verbosity`*(v[, force]) | |
| *`system_info.get_info`*(name[, notfound_action]) | notfound_action: |
| *`system_info.get_standard_file`*(fname) | Returns a list of files named 'fname' from 1) System-wide directory (directory-location of this module) 2) Users HOME directory (os.environ['HOME']) 3) Local directory |

### Functions

| | |
|---|---|
| `CCompiler_compile`(self, sources[, …]) | Compile one or more source files. |
| `CCompiler_customize`(self, dist[, need_cxx]) | Do any platform-specific customization of a compiler instance. |
| `CCompiler_customize_cmd`(self, cmd[, ignore]) | Customize compiler using distutils command. |
| `CCompiler_cxx_compiler`(self) | Return the C++ compiler. |
| `CCompiler_find_executables`(self) | Does nothing here, but is called by the get_version method and can be overridden by subclasses. |
| `CCompiler_get_version`(self[, force, ok_status]) | Return compiler version, or None if compiler is not available. |
| `CCompiler_object_filenames`(self, …[, …]) | Return the name of the object files for the given source files. |
| `CCompiler_show_customization`(self) | Print the compiler customizations to stdout. |
| `CCompiler_spawn`(self, cmd[, display]) | Execute a command in a sub-process. |
| `gen_lib_options`(compiler, library_dirs, …) | |
| `new_compiler`([plat, compiler, verbose, …]) | |
| `replace_method`(klass, method_name, func) | |
| `simple_version_match`([pat, ignore, start]) | Simple matching of version numbers, for use in CCompiler and FCompiler. |

`numpy.distutils.cpuinfo.`**`cpu = <numpy.distutils.cpuinfo.LinuxCPUInfo object>`**

**`class`** `numpy.distutils.core.`**`Extension`**(*name*, *sources*, *include_dirs=None*, *define_macros=None*, *undef_macros=None*, *library_dirs=None*, *libraries=None*, *runtime_library_dirs=None*, *extra_objects=None*, *extra_compile_args=None*, *extra_link_args=None*, *export_symbols=None*, *swig_opts=None*, *depends=None*, *language=None*, *f2py_options=None*, *module_dirs=None*, *extra_f77_compile_args=None*, *extra_f90_compile_args=None*)

**Parameters**

**name**

[str] Extension name.

**sources**

[list of str] List of source file locations relative to the top directory of the package.

**extra_compile_args**

[list of str] Extra command line arguments to pass to the compiler.

**extra_f77_compile_args**

[list of str] Extra command line arguments to pass to the fortran77 compiler.

**extra_f90_compile_args**

[list of str] Extra command line arguments to pass to the fortran90 compiler.

### Methods

| | |
|---|---|
| **has_cxx_sources** | |
| **has_f2py_sources** | |

exec_command

Implements exec_command function that is (almost) equivalent to commands.getstatusoutput function but on NT, DOS systems the returned status is actually correct (though, the returned status values may be different by a factor). In addition, exec_command takes keyword arguments for (re-)defining environment variables.

Provides functions:

**exec_command — execute command in a specified directory and**

in the modified environment.

**find_executable — locate a command using info from environment**

variable PATH. Equivalent to posix *which* command.

Author: Pearu Peterson <pearu@cens.ioc.ee> Created: 11 January 2003

Requires: Python 2.x

Successfully tested on:

| os.name | sys.platform | comments |
|---|---|---|
| posix | linux2 | Debian (sid) Linux, Python 2.1.3+, 2.2.3+, 2.3.3 PyCrust 0.9.3, Idle 1.0.2 |
| posix | linux2 | Red Hat 9 Linux, Python 2.1.3, 2.2.2, 2.3.2 |
| posix | sunos5 | SunOS 5.9, Python 2.2, 2.3.2 |
| posix | darwin | Darwin 7.2.0, Python 2.3 |
| nt | win32 | Windows Me Python 2.3(EE), Idle 1.0, PyCrust 0.7.2 Python 2.1.1 Idle 0.8 |
| nt | win32 | Windows 98, Python 2.1.1. Idle 0.8 |
| nt | win32 | Cygwin 98-4.10, Python 2.1.1(MSC) - echo tests fail i.e. redefining environment variables may not work. FIXED: don’t use cygwin echo! Comment: also *cmd /c echo* will not work but redefining environment variables do work. |
| posix | cygwin | Cygwin 98-4.10, Python 2.3.3(cygming special) |
| nt | win32 | Windows XP, Python 2.3.3 |

Known bugs:

- Tests, that send messages to stderr, fail when executed from MSYS prompt because the messages are lost at some point.

### Functions

| | |
|---|---|
| `exec_command`(command[, execute_in, …]) | Return (status,output) of executed command. |
| `filepath_from_subprocess_output`(output) | Convert *bytes* in the encoding used by a subprocess into a filesystem-appropriate *str*. |
| `find_executable`(exe[, path, _cache]) | Return full path of a executable or None. |
| `forward_bytes_to_stdout`(val) | Forward bytes from a subprocess call to the console, without attempting to decode them. |
| `get_pythonexe`() | |
| `temp_file_name`() | |

`numpy.distutils.log.`**`set_verbosity`**(*v*, *force=False*)

`numpy.distutils.system_info.`**`get_info`**(*name*, *notfound_action=0*)

**notfound_action:**

0 - do nothing 1 - display warning message 2 - raise error

`numpy.distutils.system_info.`**`get_standard_file`**(*fname*)

Returns a list of files named 'fname' from 1) System-wide directory (directory-location of this module) 2) Users HOME directory (os.environ['HOME']) 3) Local directory

## 6.2 Configuration class

**`class`** `numpy.distutils.misc_util.`**`Configuration`**(*package_name=None*, *parent_name=None*, *top_path=None*, *package_path=None*, ***attrs*)

Construct a configuration instance for the given package name. If *parent_name* is not None, then construct the package as a sub-package of the *parent_name* package. If *top_path* and *package_path* are None then they are assumed equal to the path of the file this instance was created in. The setup.py files in the numpy distribution are good examples of how to use the `Configuration` instance.

**`todict`**(*self*)

Return a dictionary compatible with the keyword arguments of distutils setup function.

### Examples

```
>>> setup(**config.todict())
```

**`get_distribution`**(*self*)

Return the distutils distribution object for self.

**`get_subpackage`**(*self*, *subpackage_name*, *subpackage_path=None*, *parent_name=None*, *caller_level=1*)

Return list of subpackage configurations.

**Parameters**

**subpackage_name**

[str or None] Name of the subpackage to get the configuration. '*' in subpackage_name is handled as a wildcard.

**subpackage_path**

[str] If None, then the path is assumed to be the local path plus the subpackage_name. If a setup.py file is not found in the subpackage_path, then a default configuration is used.

**parent_name**

[str] Parent name.

**add_subpackage** (*self*, *subpackage_name*, *subpackage_path=None*, *standalone=False*)

Add a sub-package to the current Configuration instance.

This is useful in a setup.py script for adding sub-packages to a package.

**Parameters**

**subpackage_name**

[str] name of the subpackage

**subpackage_path**

[str] if given, the subpackage path such as the subpackage is in subpackage_path / subpackage_name. If None,the subpackage is assumed to be located in the local path / subpackage_name.

**standalone**

[bool]

**add_data_files** (*self*, **files*)

Add data files to configuration data_files.

**Parameters**

**files**

[sequence] Argument(s) can be either

- 2-sequence (<datadir prefix>,<path to data file(s)>)
- paths to data files where python datadir prefix defaults to package dir.

### Notes

The form of each element of the files sequence is very flexible allowing many combinations of where to get the files from the package and where they should ultimately be installed on the system. The most basic usage is for an element of the files argument sequence to be a simple filename. This will cause that file from the local path to be installed to the installation path of the self.name package (package path). The file argument can also be a relative path in which case the entire relative path will be installed into the package directory. Finally, the file can be an absolute path name in which case the file will be found at the absolute path name but installed to the package path.

This basic behavior can be augmented by passing a 2-tuple in as the file argument. The first element of the tuple should specify the relative path (under the package install directory) where the remaining sequence of files should be installed to (it has nothing to do with the file-names in the source distribution). The second element of the tuple is the sequence of files that should be installed. The files in this sequence can be filenames, relative paths, or absolute paths. For absolute paths the file will be installed in the top-level package installation directory (regardless of the first argument). Filenames and relative path names will be installed in the package install directory under the path name given as the first element of the tuple.

Rules for installation paths:

1. file.txt -> (., file.txt)-> parent/file.txt
2. foo/file.txt -> (foo, foo/file.txt) -> parent/foo/file.txt
3. /foo/bar/file.txt -> (., /foo/bar/file.txt) -> parent/file.txt
4. `*`.txt -> parent/a.txt, parent/b.txt
5. foo/`*`.txt" -> parent/foo/a.txt, parent/foo/b.txt
6. `*/*.txt` -> (`*`, `*/*`.txt) -> parent/c/a.txt, parent/d/b.txt
7. (sun, file.txt) -> parent/sun/file.txt
8. (sun, bar/file.txt) -> parent/sun/file.txt
9. (sun, /foo/bar/file.txt) -> parent/sun/file.txt
10. (sun, `*`.txt) -> parent/sun/a.txt, parent/sun/b.txt
11. (sun, bar/`*`.txt) -> parent/sun/a.txt, parent/sun/b.txt
12. (sun/`*`, `*/*`.txt) -> parent/sun/c/a.txt, parent/d/b.txt

An additional feature is that the path to a data-file can actually be a function that takes no arguments and returns the actual path(s) to the data-files. This is useful when the data files are generated while building the package.

### Examples

Add files to the list of data_files to be included with the package.

```
>>> self.add_data_files('foo.dat',
...     ('fun', ['gun.dat', 'nun/pun.dat', '/tmp/sun.dat']),
...     'bar/cat.dat',
...     '/full/path/to/can.dat')
```

will install these data files to:

```
<package install directory>/
 foo.dat
 fun/
   gun.dat
   nun/
     pun.dat
 sun.dat
 bar/
   car.dat
 can.dat
```

where <package install directory> is the package (or sub-package) directory such as '/usr/lib/python2.4/site-packages/mypackage' ('C: Python2.4 Lib site-packages mypackage') or '/usr/lib/python2.4/site- packages/mypackage/mysubpackage' ('C: Python2.4 Lib site-packages mypackage mysubpackage').

**add_data_dir**(*self*, *data_path*)

Recursively add files under data_path to data_files list.

Recursively add files under data_path to the list of data_files to be installed (and distributed). The data_path can be either a relative path-name, or an absolute path-name, or a 2-tuple where the first argument shows where in the install directory the data directory should be installed to.

**Parameters**

**data_path**

[seq or str] Argument can be either

- 2-sequence (<datadir suffix>, <path to data directory>)
- path to data directory where python datadir suffix defaults to package dir.

## Notes

Rules for installation paths:

```
foo/bar -> (foo/bar, foo/bar) -> parent/foo/bar
(gun, foo/bar) -> parent/gun
foo/* -> (foo/a, foo/a), (foo/b, foo/b) -> parent/foo/a, parent/foo/b
(gun, foo/*) -> (gun, foo/a), (gun, foo/b) -> gun
(gun/*, foo/*) -> parent/gun/a, parent/gun/b
/foo/bar -> (bar, /foo/bar) -> parent/bar
(gun, /foo/bar) -> parent/gun
(fun/*/gun/*, sun/foo/bar) -> parent/fun/foo/gun/bar
```

## Examples

For example suppose the source directory contains fun/foo.dat and fun/bar/car.dat:

```
>>> self.add_data_dir('fun')
>>> self.add_data_dir(('sun', 'fun'))
>>> self.add_data_dir(('gun', '/full/path/to/fun'))
```

Will install data-files to the locations:

```
<package install directory>/
  fun/
    foo.dat
    bar/
      car.dat
  sun/
    foo.dat
    bar/
      car.dat
  gun/
    foo.dat
    car.dat
```

**`add_include_dirs`**(*self*, **paths*)

Add paths to configuration include directories.

Add the given sequence of paths to the beginning of the include_dirs list. This list will be visible to all extension modules of the current package.

**`add_headers`**(*self*, **files*)

Add installable headers to configuration.

Add the given sequence of files to the beginning of the headers list. By default, headers will be installed under <python- include>/<self.name.replace('.','/')>/ directory. If an item of files is a tuple, then its first argument specifies the actual installation location relative to the <python-include> path.

**Parameters**

**files**

[str or seq] Argument(s) can be either:

- 2-sequence (<includedir suffix>,<path to header file(s)>)
- path(s) to header file(s) where python includedir suffix will default to package name.

**add_extension** (*self*, *name*, *sources*, ***kw*)

Add extension to configuration.

Create and add an Extension instance to the ext_modules list. This method also takes the following optional keyword arguments that are passed on to the Extension constructor.

**Parameters**

**name**

[str] name of the extension

**sources**

[seq] list of the sources. The list of sources may contain functions (called source generators) which must take an extension instance and a build directory as inputs and return a source file or list of source files or None. If None is returned then no sources are generated. If the Extension instance has no sources after processing all source generators, then no extension module is built.

**include_dirs :**

**define_macros :**

**undef_macros :**

**library_dirs :**

**libraries :**

**runtime_library_dirs :**

**extra_objects :**

**extra_compile_args :**

**extra_link_args :**

**extra_f77_compile_args :**

**extra_f90_compile_args :**

**export_symbols :**

**swig_opts :**

**depends :**

The depends list contains paths to files or directories that the sources of the extension module depend on. If any path in the depends list is newer than the extension module, then the module will be rebuilt.

**language :**

**f2py_options :**

**module_dirs :**

**extra_info**

[dict or list] dict or list of dict of keywords to be appended to keywords.

### Notes

The self.paths(…) method is applied to all lists that may contain paths.

**`add_library`**(*self*, *name*, *sources*, ***build_info*)

Add library to configuration.

**Parameters**

**name**

[str] Name of the extension.

**sources**

[sequence] List of the sources. The list of sources may contain functions (called source generators) which must take an extension instance and a build directory as inputs and return a source file or list of source files or None. If None is returned then no sources are generated. If the Extension instance has no sources after processing all source generators, then no extension module is built.

**build_info**

[dict, optional] The following keys are allowed:

- depends
- macros
- include_dirs
- extra_compiler_args
- extra_f77_compile_args
- extra_f90_compile_args
- f2py_options
- language

**`add_scripts`**(*self*, **files*)

Add scripts to configuration.

Add the sequence of files to the beginning of the scripts list. Scripts will be installed under the <prefix>/bin/ directory.

**`add_installed_library`**(*self*, *name*, *sources*, *install_dir*, *build_info=None*)

Similar to add_library, but the specified library is installed.

Most C libraries used with `distutils` are only used to build python extensions, but libraries built through this method will be installed so that they can be reused by third-party packages.

**Parameters**

**name**

[str] Name of the installed library.

**sources**

[sequence] List of the library's source files. See *add_library* for details.

**install_dir**

[str] Path to install the library, relative to the current sub-package.

**build_info**

[dict, optional] The following keys are allowed:

- depends
- macros
- include_dirs
- extra_compiler_args
- extra_f77_compile_args
- extra_f90_compile_args
- f2py_options
- language

**Returns**

**None**

**See also:**

*add_library*, *add_npy_pkg_config*, *get_info*

### Notes

The best way to encode the options required to link against the specified C libraries is to use a "libname.ini" file, and use *get_info* to retrieve the required options (see *add_npy_pkg_config* for more information).

**add_npy_pkg_config**(*self*, *template*, *install_dir*, *subst_dict=None*)

Generate and install a npy-pkg config file from a template.

The config file generated from *template* is installed in the given install directory, using *subst_dict* for variable substitution.

**Parameters**

**template**

[str] The path of the template, relatively to the current package path.

**install_dir**

[str] Where to install the npy-pkg config file, relatively to the current package path.

**subst_dict**

[dict, optional] If given, any string of the form `@key@` will be replaced by `subst_dict[key]` in the template file when installed. The install prefix is always available through the variable `@prefix@`, since the install prefix is not easy to get reliably from setup.py.

**See also:**

*add_installed_library*, *get_info*

## Notes

This works for both standard installs and in-place builds, i.e. the `@prefix@` refer to the source directory for in-place builds.

## Examples

```
config.add_npy_pkg_config('foo.ini.in', 'lib', {'foo': bar})
```

Assuming the foo.ini.in file has the following content:

```
[meta]
Name=@foo@
Version=1.0
Description=dummy description

[default]
Cflags=-I@prefix@/include
Libs=
```

The generated file will have the following content:

```
[meta]
Name=bar
Version=1.0
Description=dummy description

[default]
Cflags=-Iprefix_dir/include
Libs=
```

and will be installed as foo.ini in the 'lib' subpath.

When cross-compiling with numpy distutils, it might be necessary to use modified npy-pkg-config files. Using the default/generated files will link with the host libraries (i.e. libnpymath.a). For cross-compilation you of-course need to link with target libraries, while using the host Python installation.

You can copy out the numpy/core/lib/npy-pkg-config directory, add a pkgdir value to the .ini files and set NPY_PKG_CONFIG_PATH environment variable to point to the directory with the modified npy-pkg-config files.

Example npymath.ini modified for cross-compilation:

```
[meta]
Name=npymath
Description=Portable, core math library implementing C99 standard
Version=0.1

[variables]
pkgname=numpy.core
pkgdir=/build/arm-linux-gnueabi/sysroot/usr/lib/python3.7/site-packages/numpy/
↪core
```

(continues on next page)

(continued from previous page)

```
prefix=${pkgdir}
libdir=${prefix}/lib
includedir=${prefix}/include

[default]
Libs=-L${libdir} -lnpymath
Cflags=-I${includedir}
Requires=mlib

[msvc]
Libs=/LIBPATH:${libdir} npymath.lib
Cflags=/INCLUDE:${includedir}
Requires=mlib
```

**paths**(*self*, **paths*, ***kws*)

Apply glob to paths and prepend local_path if needed.

Applies glob.glob(…) to each path in the sequence (if needed) and pre-pends the local_path if needed. Because this is called on all source lists, this allows wildcard characters to be specified in lists of sources for extension modules and libraries and scripts and allows path-names be relative to the source directory.

**get_config_cmd**(*self*)

Returns the numpy.distutils config command instance.

**get_build_temp_dir**(*self*)

Return a path to a temporary directory where temporary files should be placed.

**have_f77c**(*self*)

Check for availability of Fortran 77 compiler.

Use it inside source generating function to ensure that setup distribution instance has been initialized.

### Notes

True if a Fortran 77 compiler is available (because a simple Fortran 77 code was able to be compiled successfully).

**have_f90c**(*self*)

Check for availability of Fortran 90 compiler.

Use it inside source generating function to ensure that setup distribution instance has been initialized.

### Notes

True if a Fortran 90 compiler is available (because a simple Fortran 90 code was able to be compiled successfully)

**get_version**(*self*, *version_file=None*, *version_variable=None*)

Try to get version string of a package.

Return a version string of the current package or None if the version information could not be detected.

### Notes

This method scans files named __version__.py, <packagename>_version.py, version.py, and __svn_version__.py for string variables version, __version__, and <packagename>_version, until a version number is found.

**`make_svn_version_py`**(*self*, *delete=True*)

Appends a data function to the data_files list that will generate __svn_version__.py file to the current package directory.

Generate package __svn_version__.py file from SVN revision number, it will be removed after python exits but will be available when sdist, etc commands are executed.

### Notes

If __svn_version__.py existed before, nothing is done.

This is intended for working with source directories that are in an SVN repository.

**`make_config_py`**(*self*, *name='__config__'*)

Generate package __config__.py file containing system_info information used during building the package.

This file is installed to the package installation directory.

**`get_info`**(*self*, **names*)

Get resources information.

Return information (from system_info.get_info) for all of the names in the argument list in a single dictionary.

## 6.3 Building Installable C libraries

Conventional C libraries (installed through *add_library*) are not installed, and are just used during the build (they are statically linked). An installable C library is a pure C library, which does not depend on the python C runtime, and is installed such that it may be used by third-party packages. To build and install the C library, you just use the method *add_installed_library* instead of *add_library*, which takes the same arguments except for an additional `install_dir` argument:

```
.. hidden in a comment so as to be included in refguide but not rendered documentation
  >>> import numpy.distutils.misc_util
  >>> config = np.distutils.misc_util.Configuration(None, '', '.')
  >>> with open('foo.c', 'w') as f: pass

>>> config.add_installed_library('foo', sources=['foo.c'], install_dir='lib')
```

### 6.3.1 npy-pkg-config files

To make the necessary build options available to third parties, you could use the *npy-pkg-config* mechanism implemented in *`numpy.distutils`*. This mechanism is based on a .ini file which contains all the options. A .ini file is very similar to .pc files as used by the pkg-config unix utility:

```
[meta]
Name: foo
Version: 1.0
Description: foo library
```

(continues on next page)

(continued from previous page)

```
[variables]
prefix = /home/user/local
libdir = ${prefix}/lib
includedir = ${prefix}/include

[default]
cflags = -I${includedir}
libs = -L${libdir} -lfoo
```

Generally, the file needs to be generated during the build, since it needs some information known at build time only (e.g. prefix). This is mostly automatic if one uses the `Configuration` method *add_npy_pkg_config*. Assuming we have a template file foo.ini.in as follows:

```
[meta]
Name: foo
Version: @version@
Description: foo library

[variables]
prefix = @prefix@
libdir = ${prefix}/lib
includedir = ${prefix}/include

[default]
cflags = -I${includedir}
libs = -L${libdir} -lfoo
```

and the following code in setup.py:

```
>>> config.add_installed_library('foo', sources=['foo.c'], install_dir='lib')
>>> subst = {'version': '1.0'}
>>> config.add_npy_pkg_config('foo.ini.in', 'lib', subst_dict=subst)
```

This will install the file foo.ini into the directory package_dir/lib, and the foo.ini file will be generated from foo.ini.in, where each `@version@` will be replaced by `subst_dict['version']`. The dictionary has an additional prefix substitution rule automatically added, which contains the install prefix (since this is not easy to get from setup.py). npy-pkg-config files can also be installed at the same location as used for numpy, using the path returned from `get_npy_pkg_dir` function.

### 6.3.2 Reusing a C library from another package

Info are easily retrieved from the `get_info` function in `numpy.distutils.misc_util`:

```
>>> info = np.distutils.misc_util.get_info('npymath')
>>> config.add_extension('foo', sources=['foo.c'], extra_info=info)
```

An additional list of paths to look for .ini files can be given to `get_info`.

## 6.4 Conversion of `.src` files

NumPy distutils supports automatic conversion of source files named <somefile>.src. This facility can be used to maintain very similar code blocks requiring only simple changes between blocks. During the build phase of setup, if a template file named <somefile>.src is encountered, a new file named <somefile> is constructed from the template and placed in the build directory to be used instead. Two forms of template conversion are supported. The first form occurs for files named <file>.ext.src where ext is a recognized Fortran extension (f, f90, f95, f77, for, ftn, pyf). The second form is used for all other cases. See *Conversion of .src files using Templates*.

CHAPTER

SEVEN

# NUMPY DISTUTILS - USERS GUIDE

## 7.1 SciPy structure

Currently SciPy project consists of two packages:

- NumPy — it provides packages like:
  - numpy.distutils - extension to Python distutils
  - numpy.f2py - a tool to bind Fortran/C codes to Python
  - numpy.core - future replacement of Numeric and numarray packages
  - numpy.lib - extra utility functions
  - numpy.testing - numpy-style tools for unit testing
  - etc
- SciPy — a collection of scientific tools for Python.

The aim of this document is to describe how to add new tools to SciPy.

## 7.2 Requirements for SciPy packages

SciPy consists of Python packages, called SciPy packages, that are available to Python users via the `scipy` namespace. Each SciPy package may contain other SciPy packages. And so on. Therefore, the SciPy directory tree is a tree of packages with arbitrary depth and width. Any SciPy package may depend on NumPy packages but the dependence on other SciPy packages should be kept minimal or zero.

A SciPy package contains, in addition to its sources, the following files and directories:

- `setup.py` — building script
- `__init__.py` — package initializer
- `tests/` — directory of unittests

Their contents are described below.

## 7.3 The `setup.py` file

In order to add a Python package to SciPy, its build script (`setup.py`) must meet certain requirements. The most important requirement is that the package define a `configuration(parent_package='',top_path=None)` function which returns a dictionary suitable for passing to `numpy.distutils.core.setup(..)`. To simplify the construction of this dictionary, `numpy.distutils.misc_util` provides the `Configuration` class, described below.

### 7.3.1 SciPy pure Python package example

Below is an example of a minimal `setup.py` file for a pure SciPy package:

```
#!/usr/bin/env python3
def configuration(parent_package='',top_path=None):
    from numpy.distutils.misc_util import Configuration
    config = Configuration('mypackage',parent_package,top_path)
    return config

if __name__ == "__main__":
    from numpy.distutils.core import setup
    #setup(**configuration(top_path='').todict())
    setup(configuration=configuration)
```

The arguments of the `configuration` function specify the name of parent SciPy package (`parent_package`) and the directory location of the main `setup.py` script (`top_path`). These arguments, along with the name of the current package, should be passed to the `Configuration` constructor.

The `Configuration` constructor has a fourth optional argument, `package_path`, that can be used when package files are located in a different location than the directory of the `setup.py` file.

Remaining `Configuration` arguments are all keyword arguments that will be used to initialize attributes of `Configuration` instance. Usually, these keywords are the same as the ones that `setup(..)` function would expect, for example, `packages`, `ext_modules`, `data_files`, `include_dirs`, `libraries`, `headers`, `scripts`, `package_dir`, etc. However, the direct specification of these keywords is not recommended as the content of these keyword arguments will not be processed or checked for the consistency of SciPy building system.

Finally, `Configuration` has `.todict()` method that returns all the configuration data as a dictionary suitable for passing on to the `setup(..)` function.

### 7.3.2 `Configuration` instance attributes

In addition to attributes that can be specified via keyword arguments to `Configuration` constructor, `Configuration` instance (let us denote as `config`) has the following attributes that can be useful in writing setup scripts:

- `config.name` - full name of the current package. The names of parent packages can be extracted as `config.name.split('.')`.
- `config.local_path` - path to the location of current `setup.py` file.
- `config.top_path` - path to the location of main `setup.py` file.

### 7.3.3 `Configuration` instance methods

- `config.todict()` — returns configuration dictionary suitable for passing to `numpy.distutils.core.setup(..)` function.
- `config.paths(*paths) --- applies ``glob.glob(..)` to items of `paths` if necessary. Fixes `paths` item that is relative to `config.local_path`.
- `config.get_subpackage(subpackage_name,subpackage_path=None)` — returns a list of subpackage configurations. Subpackage is looked in the current directory under the name `subpackage_name` but the path can be specified also via optional `subpackage_path` argument. If `subpackage_name` is specified as `None` then the subpackage name will be taken the basename of `subpackage_path`. Any `*` used for subpackage names are expanded as wildcards.
- `config.add_subpackage(subpackage_name,subpackage_path=None)` — add SciPy subpackage configuration to the current one. The meaning and usage of arguments is explained above, see `config.get_subpackage()` method.
- `config.add_data_files(*files)` — prepend `files` to `data_files` list. If `files` item is a tuple then its first element defines the suffix of where data files are copied relative to package installation directory and the second element specifies the path to data files. By default data files are copied under package installation directory. For example,

  ```
  config.add_data_files('foo.dat',
                        ('fun',['gun.dat','nun/pun.dat','/tmp/sun.dat']),
                        'bar/car.dat'.
                        '/full/path/to/can.dat',
                        )
  ```

  will install data files to the following locations

  ```
  <installation path of config.name package>/
    foo.dat
    fun/
      gun.dat
      pun.dat
      sun.dat
    bar/
      car.dat
    can.dat
  ```

  Path to data files can be a function taking no arguments and returning path(s) to data files – this is a useful when data files are generated while building the package. (XXX: explain the step when this function are called exactly)

- `config.add_data_dir(data_path)` — add directory `data_path` recursively to `data_files`. The whole directory tree starting at `data_path` will be copied under package installation directory. If `data_path` is a tuple then its first element defines the suffix of where data files are copied relative to package installation directory and the second element specifies the path to data directory. By default, data directory are copied under package installation directory under the basename of `data_path`. For example,

  ```
  config.add_data_dir('fun')  # fun/ contains foo.dat bar/car.dat
  config.add_data_dir(('sun','fun'))
  config.add_data_dir(('gun','/full/path/to/fun'))
  ```

  will install data files to the following locations

  ```
  <installation path of config.name package>/
    fun/
  ```

(continues on next page)

(continued from previous page)

```
      foo.dat
      bar/
         car.dat
   sun/
      foo.dat
      bar/
         car.dat
   gun/
      foo.dat
      bar/
         car.dat
```

- `config.add_include_dirs(*paths)` — prepend `paths` to `include_dirs` list. This list will be visible to all extension modules of the current package.
- `config.add_headers(*files)` — prepend `files` to `headers` list. By default, headers will be installed under `<prefix>/include/pythonX.X/<config.name.replace('.','/')>/` directory. If `files` item is a tuple then it's first argument specifies the installation suffix relative to `<prefix>/include/pythonX.X/` path. This is a Python distutils method; its use is discouraged for NumPy and SciPy in favour of `config.add_data_files(*files)`.
- `config.add_scripts(*files)` — prepend `files` to `scripts` list. Scripts will be installed under `<prefix>/bin/` directory.
- `config.add_extension(name,sources,**kw)` — create and add an `Extension` instance to `ext_modules` list. The first argument `name` defines the name of the extension module that will be installed under `config.name` package. The second argument is a list of sources. `add_extension` method takes also keyword arguments that are passed on to the `Extension` constructor. The list of allowed keywords is the following: `include_dirs`, `define_macros`, `undef_macros`, `library_dirs`, `libraries`, `runtime_library_dirs`, `extra_objects`, `extra_compile_args`, `extra_link_args`, `export_symbols`, `swig_opts`, `depends`, `language`, `f2py_options`, `module_dirs`, `extra_info`, `extra_f77_compile_args`, `extra_f90_compile_args`.

  Note that `config.paths` method is applied to all lists that may contain paths. `extra_info` is a dictionary or a list of dictionaries that content will be appended to keyword arguments. The list `depends` contains paths to files or directories that the sources of the extension module depend on. If any path in the `depends` list is newer than the extension module, then the module will be rebuilt.

  The list of sources may contain functions ('source generators') with a pattern `def <funcname>(ext, build_dir): return <source(s) or None>`. If `funcname` returns `None`, no sources are generated. And if the `Extension` instance has no sources after processing all source generators, no extension module will be built. This is the recommended way to conditionally define extension modules. Source generator functions are called by the `build_src` sub-command of `numpy.distutils`.

  For example, here is a typical source generator function:

  ```python
  def generate_source(ext,build_dir):
      import os
      from distutils.dep_util import newer
      target = os.path.join(build_dir,'somesource.c')
      if newer(target,__file__):
          # create target file
      return target
  ```

  The first argument contains the Extension instance that can be useful to access its attributes like `depends`, `sources`, etc. lists and modify them during the building process. The second argument gives a path to a build directory that must be used when creating files to a disk.

- `config.add_library(name, sources, **build_info)` — add a library to `libraries` list. Allowed keywords arguments are `depends`, `macros`, `include_dirs`, `extra_compiler_args`, `f2py_options`, `extra_f77_compile_args`, `extra_f90_compile_args`. See . `add_extension()` method for more information on arguments.
- `config.have_f77c()` — return True if Fortran 77 compiler is available (read: a simple Fortran 77 code compiled successfully).
- `config.have_f90c()` — return True if Fortran 90 compiler is available (read: a simple Fortran 90 code compiled successfully).
- `config.get_version()` — return version string of the current package, `None` if version information could not be detected. This methods scans files `__version__.py`, `<packagename>_version.py`, `version.py`, `__svn_version__.py` for string variables `version`, `__version__`, `<packagename>_version`.
- `config.make_svn_version_py()` — appends a data function to `data_files` list that will generate `__svn_version__.py` file to the current package directory. The file will be removed from the source directory when Python exits.
- `config.get_build_temp_dir()` — return a path to a temporary directory. This is the place where one should build temporary files.
- `config.get_distribution()` — return distutils `Distribution` instance.
- `config.get_config_cmd()` — returns `numpy.distutils` config command instance.
- `config.get_info(*names)` —

### 7.3.4 Conversion of `.src` files using Templates

NumPy distutils supports automatic conversion of source files named <somefile>.src. This facility can be used to maintain very similar code blocks requiring only simple changes between blocks. During the build phase of setup, if a template file named <somefile>.src is encountered, a new file named <somefile> is constructed from the template and placed in the build directory to be used instead. Two forms of template conversion are supported. The first form occurs for files named <file>.ext.src where ext is a recognized Fortran extension (f, f90, f95, f77, for, ftn, pyf). The second form is used for all other cases.

### 7.3.5 Fortran files

This template converter will replicate all **function** and **subroutine** blocks in the file with names that contain ‘<…>’ according to the rules in ‘<…>’. The number of comma-separated words in ‘<…>’ determines the number of times the block is repeated. What these words are indicates what that repeat rule, ‘<…>’, should be replaced with in each block. All of the repeat rules in a block must contain the same number of comma-separated words indicating the number of times that block should be repeated. If the word in the repeat rule needs a comma, leftarrow, or rightarrow, then prepend it with a backslash ‘ ‘. If a word in the repeat rule matches ‘ \<index>’ then it will be replaced with the <index>-th word in the same repeat specification. There are two forms for the repeat rule: named and short.

### Named repeat rule

A named repeat rule is useful when the same set of repeats must be used several times in a block. It is specified using <rule1=item1, item2, item3,…, itemN>, where N is the number of times the block should be repeated. On each repeat of the block, the entire expression, '<…>' will be replaced first with item1, and then with item2, and so forth until N repeats are accomplished. Once a named repeat specification has been introduced, the same repeat rule may be used **in the current block** by referring only to the name (i.e. <rule1>).

### Short repeat rule

A short repeat rule looks like <item1, item2, item3, …, itemN>. The rule specifies that the entire expression, '<…>' should be replaced first with item1, and then with item2, and so forth until N repeats are accomplished.

### Pre-defined names

The following predefined named repeat rules are available:

- <prefix=s,d,c,z>
- <_c=s,d,c,z>
- <_t=real, double precision, complex, double complex>
- <ftype=real, double precision, complex, double complex>
- <ctype=float, double, complex_float, complex_double>
- <ftypereal=float, double precision, \0, \1>
- <ctypereal=float, double, \0, \1>

## 7.3.6 Other files

Non-Fortran files use a separate syntax for defining template blocks that should be repeated using a variable expansion similar to the named repeat rules of the Fortran-specific repeats.

NumPy Distutils preprocesses C source files (extension: `.c.src`) written in a custom templating language to generate C code. The `@` symbol is used to wrap macro-style variables to empower a string substitution mechanism that might describe (for instance) a set of data types.

The template language blocks are delimited by `/**begin repeat` and `/**end repeat**/` lines, which may also be nested using consecutively numbered delimiting lines such as `/**begin repeat1` and `/**end repeat1**/`:

1. "/**begin repeat "on a line by itself marks the beginning of a segment that should be repeated.

2. Named variable expansions are defined using `#name=item1, item2, item3, ..., itemN#` and placed on successive lines. These variables are replaced in each repeat block with corresponding word. All named variables in the same repeat block must define the same number of words.

3. In specifying the repeat rule for a named variable, `item*N` is short- hand for `item, item, ..., item` repeated N times. In addition, parenthesis in combination with *N can be used for grouping several items that should be repeated. Thus, #name=(item1, item2)*4# is equivalent to #name=item1, item2, item1, item2, item1, item2, item1, item2#

4. "*/ "on a line by itself marks the end of the variable expansion naming. The next line is the first line that will be repeated using the named rules.

5. Inside the block to be repeated, the variables that should be expanded are specified as `@name@`

6. "/**end repeat**/ "on a line by itself marks the previous line as the last line of the block to be repeated.

7. A loop in the NumPy C source code may have a `@TYPE@` variable, targeted for string substitution, which is preprocessed to a number of otherwise identical loops with several strings such as INT, LONG, UINT, ULONG. The `@TYPE@` style syntax thus reduces code duplication and maintenance burden by mimicking languages that have generic type support.

The above rules may be clearer in the following template source example:

```
/* TIMEDELTA to non-float types */

/**begin repeat
 *
 * #TOTYPE = BYTE, UBYTE, SHORT, USHORT, INT, UINT, LONG, ULONG,
 *           LONGLONG, ULONGLONG, DATETIME,
 *           TIMEDELTA#
 * #totype = npy_byte, npy_ubyte, npy_short, npy_ushort, npy_int, npy_uint,
 *           npy_long, npy_ulong, npy_longlong, npy_ulonglong,
 *           npy_datetime, npy_timedelta#
 */

/**begin repeat1
 *
 * #FROMTYPE = TIMEDELTA#
 * #fromtype = npy_timedelta#
 */
static void
@FROMTYPE@_to_@TOTYPE@(void *input, void *output, npy_intp n,
        void *NPY_UNUSED(aip), void *NPY_UNUSED(aop))
{
    const @fromtype@ *ip = input;
    @totype@ *op = output;

    while (n--) {
        *op++ = (@totype@)*ip++;
    }
}
/**end repeat1**/

/**end repeat**/
```

The preprocessing of generically typed C source files (whether in NumPy proper or in any third party package using NumPy Distutils) is performed by conv_template.py. The type specific C files generated (extension: .c) by these modules during the build process are ready to be compiled. This form of generic typing is also supported for C header files (preprocessed to produce .h files).

### 7.3.7 Useful functions in `numpy.distutils.misc_util`

- `get_numpy_include_dirs()` — return a list of NumPy base include directories. NumPy base include directories contain header files such as `numpy/arrayobject.h`, `numpy/funcobject.h` etc. For installed NumPy the returned list has length 1 but when building NumPy the list may contain more directories, for example, a path to `config.h` file that `numpy/base/setup.py` file generates and is used by `numpy` header files.
- `append_path(prefix,path)` — smart append `path` to `prefix`.
- `gpaths(paths, local_path='')` — apply glob to paths and prepend `local_path` if needed.
- `njoin(*path)` — join pathname components + convert `/`-separated path to `os.sep`-separated path and resolve `..`, `.` from paths. Ex. `njoin('a',['b','./c'],'..','g') -> os.path.join('a', 'b','g')`.

- `minrelpath(path)` — resolves dots in `path`.
- `rel_path(path, parent_path)` — return `path` relative to `parent_path`.
- `def get_cmd(cmdname,_cache={})` — returns `numpy.distutils` command instance.
- `all_strings(lst)`
- `has_f_sources(sources)`
- `has_cxx_sources(sources)`
- `filter_sources(sources)` — return `c_sources, cxx_sources, f_sources, fmodule_sources`
- `get_dependencies(sources)`
- `is_local_src_dir(directory)`
- `get_ext_source_files(ext)`
- `get_script_files(scripts)`
- `get_lib_source_files(lib)`
- `get_data_files(data)`
- `dot_join(*args)` — join non-zero arguments with a dot.
- `get_frame(level=0)` — return frame object from call stack with given level.
- `cyg2win32(path)`
- `mingw32()` — return `True` when using mingw32 environment.
- `terminal_has_colors()`, `red_text(s)`, `green_text(s)`, `yellow_text(s)`, `blue_text(s)`, `cyan_text(s)`
- `get_path(mod_name,parent_path=None)` — return path of a module relative to parent_path when given. Handles also `__main__` and `__builtin__` modules.
- `allpath(name)` — replaces `/` with `os.sep` in `name`.
- `cxx_ext_match`, `fortran_ext_match`, `f90_ext_match`, `f90_module_name_match`

### 7.3.8 `numpy.distutils.system_info` module

- `get_info(name,notfound_action=0)`
- `combine_paths(*args,**kws)`
- `show_all()`

### 7.3.9 `numpy.distutils.cpuinfo` module

- `cpuinfo`

### 7.3.10 `numpy.distutils.log` module

- `set_verbosity(v)`

### 7.3.11 `numpy.distutils.exec_command` module

- `get_pythonexe()`
- `find_executable(exe, path=None)`
- `exec_command( command, execute_in='', use_shell=None, use_tee=None, **env )`

## 7.4 The `__init__.py` file

The header of a typical SciPy `__init__.py` is:

```
"""
Package docstring, typically with a brief description and function listing.
"""

# import functions into module namespace
from .subpackage import *
...

__all__ = [s for s in dir() if not s.startswith('_')]

from numpy.testing import Tester
test = Tester().test
bench = Tester().bench
```

Note that NumPy submodules still use a file named `info.py` in which the module docstring and `__all__` dict are defined. These files will be removed at some point.

## 7.5 Extra features in NumPy Distutils

### 7.5.1 Specifying config_fc options for libraries in setup.py script

It is possible to specify config_fc options in setup.py scripts. For example, using

> **config.add_library('library',**
>
> > sources=[…], config_fc={'noopt':(__file__,1)})

will compile the `library` sources without optimization flags.

It's recommended to specify only those config_fc options in such a way that are compiler independent.

### 7.5.2 Getting extra Fortran 77 compiler options from source

Some old Fortran codes need special compiler options in order to work correctly. In order to specify compiler options per source file, `numpy.distutils` Fortran compiler looks for the following pattern:

```
CF77FLAGS(<fcompiler type>) = <fcompiler f77flags>
```

in the first 20 lines of the source and use the `f77flags` for specified type of the fcompiler (the first character `C` is optional).

TODO: This feature can be easily extended for Fortran 90 codes as well. Let us know if you would need such a feature.

## CHAPTER

## EIGHT

# NUMPY C-API

Beware of the man who won't be bothered with details.
— *William Feather, Sr.*

The truth is out there.
— *Chris Carter, The X Files*

NumPy provides a C-API to enable users to extend the system and get access to the array object for use in other routines. The best way to truly understand the C-API is to read the source code. If you are unfamiliar with (C) source code, however, this can be a daunting experience at first. Be assured that the task becomes easier with practice, and you may be surprised at how simple the C-code can be to understand. Even if you don't think you can write C-code from scratch, it is much easier to understand and modify already-written source code than create it *de novo*.

Python extensions are especially straightforward to understand because they all have a very similar structure. Admittedly, NumPy is not a trivial extension to Python, and may take a little more snooping to grasp. This is especially true because of the code-generation techniques, which simplify maintenance of very similar code, but can make the code a little less readable to beginners. Still, with a little persistence, the code can be opened to your understanding. It is my hope, that this guide to the C-API can assist in the process of becoming familiar with the compiled-level work that can be done with NumPy in order to squeeze that last bit of necessary speed out of your code.

## 8.1 Python Types and C-Structures

Several new types are defined in the C-code. Most of these are accessible from Python, but a few are not exposed due to their limited use. Every new Python type has an associated `PyObject *` with an internal structure that includes a pointer to a "method table" that defines how the new object behaves in Python. When you receive a Python object into C code, you always get a pointer to a `PyObject` structure. Because a `PyObject` structure is very generic and defines only `PyObject_HEAD`, by itself it is not very interesting. However, different objects contain more details after the `PyObject_HEAD` (but you have to cast to the correct type to access them — or use accessor functions or macros).

## 8.1.1 New Python Types Defined

Python types are the functional equivalent in C of classes in Python. By constructing a new Python type you make available a new object for Python. The ndarray object is an example of a new type defined in C. New types are defined in C by two basic steps:

1. creating a C-structure (usually named `Py{Name}Object`) that is binary- compatible with the `PyObject` structure itself but holds the additional information needed for that particular object;
2. populating the `PyTypeObject` table (pointed to by the ob_type member of the `PyObject` structure) with pointers to functions that implement the desired behavior for the type.

Instead of special method names which define behavior for Python classes, there are "function tables" which point to functions that implement the desired results. Since Python 2.2, the PyTypeObject itself has become dynamic which allows C types that can be "sub-typed "from other C-types in C, and sub-classed in Python. The children types inherit the attributes and methods from their parent(s).

There are two major new types: the ndarray ( *`PyArray_Type`* ) and the ufunc ( *`PyUFunc_Type`* ). Additional types play a supportive role: the *`PyArrayIter_Type`*, the *`PyArrayMultiIter_Type`*, and the *`PyArrayDescr_Type`* . The *`PyArrayIter_Type`* is the type for a flat iterator for an ndarray (the object that is returned when getting the flat attribute). The *`PyArrayMultiIter_Type`* is the type of the object returned when calling `broadcast` (). It handles iteration and broadcasting over a collection of nested sequences. Also, the *`PyArrayDescr_Type`* is the data-type-descriptor type whose instances describe the data. Finally, there are 21 new scalar-array types which are new Python scalars corresponding to each of the fundamental data types available for arrays. An additional 10 other types are place holders that allow the array scalars to fit into a hierarchy of actual Python types.

### PyArray_Type and PyArrayObject

**`PyArray_Type`**

The Python type of the ndarray is *`PyArray_Type`*. In C, every ndarray is a pointer to a *`PyArrayObject`* structure. The ob_type member of this structure contains a pointer to the *`PyArray_Type`* typeobject.

**`PyArrayObject`**

The *`PyArrayObject`* C-structure contains all of the required information for an array. All instances of an ndarray (and its subclasses) will have this structure. For future compatibility, these structure members should normally be accessed using the provided macros. If you need a shorter name, then you can make use of `NPY_AO` (deprecated) which is defined to be equivalent to *`PyArrayObject`*. Direct access to the struct fields are deprecated. Use the *PyArray_*(arr)* form instead.

```
typedef struct PyArrayObject {
    PyObject_HEAD
    char *data;
    int nd;
    npy_intp *dimensions;
    npy_intp *strides;
    PyObject *base;
    PyArray_Descr *descr;
    int flags;
    PyObject *weakreflist;
} PyArrayObject;
```

**`PyArrayObject.PyObject_HEAD`**

This is needed by all Python objects. It consists of (at least) a reference count member ( `ob_refcnt` ) and a pointer to the typeobject ( `ob_type` ). (Other elements may also be present if Python was compiled with special options see Include/object.h in the Python source tree for more information). The ob_type member points to a Python type object.

char ***PyArrayObject.data**

Accessible via *PyArray_DATA*, this data member is a pointer to the first element of the array. This pointer can (and normally should) be recast to the data type of the array.

int **PyArrayObject.nd**

An integer providing the number of dimensions for this array. When nd is 0, the array is sometimes called a rank-0 array. Such arrays have undefined dimensions and strides and cannot be accessed. Macro *PyArray_NDIM* defined in `ndarraytypes.h` points to this data member. *NPY_MAXDIMS* is the largest number of dimensions for any array.

*npy_intp* **PyArrayObject.dimensions**

An array of integers providing the shape in each dimension as long as nd $\geq 1$. The integer is always large enough to hold a pointer on the platform, so the dimension size is only limited by memory. *PyArray_DIMS* is the macro associated with this data member.

*npy_intp* ***PyArrayObject.strides**

An array of integers providing for each dimension the number of bytes that must be skipped to get to the next element in that dimension. Associated with macro *PyArray_STRIDES*.

PyObject ***PyArrayObject.base**

Pointed to by *PyArray_BASE*, this member is used to hold a pointer to another Python object that is related to this array. There are two use cases:

- If this array does not own its own memory, then base points to the Python object that owns it (perhaps another array object)
- If this array has the (deprecated) *NPY_ARRAY_UPDATEIFCOPY* or *NPY_ARRAY_WRITEBACKIFCOPY* flag set, then this array is a working copy of a "misbehaved" array.

When `PyArray_ResolveWritebackIfCopy` is called, the array pointed to by base will be updated with the contents of this array.

*PyArray_Descr* ***PyArrayObject.descr**

A pointer to a data-type descriptor object (see below). The data-type descriptor object is an instance of a new built-in type which allows a generic description of memory. There is a descriptor structure for each data type supported. This descriptor structure contains useful information about the type as well as a pointer to a table of function pointers to implement specific functionality. As the name suggests, it is associated with the macro *PyArray_DESCR*.

int **PyArrayObject.flags**

Pointed to by the macro *PyArray_FLAGS*, this data member represents the flags indicating how the memory pointed to by data is to be interpreted. Possible flags are *NPY_ARRAY_C_CONTIGUOUS*, *NPY_ARRAY_F_CONTIGUOUS*, *NPY_ARRAY_OWNDATA*, *NPY_ARRAY_ALIGNED*, *NPY_ARRAY_WRITEABLE*, *NPY_ARRAY_WRITEBACKIFCOPY*, and *NPY_ARRAY_UPDATEIFCOPY*.

PyObject ***PyArrayObject.weakreflist**

This member allows array objects to have weak references (using the weakref module).

### PyArrayDescr_Type and PyArray_Descr

**PyArrayDescr_Type**

The *PyArrayDescr_Type* is the built-in type of the data-type-descriptor objects used to describe how the bytes comprising the array are to be interpreted. There are 21 statically-defined *PyArray_Descr* objects for the built-in data-types. While these participate in reference counting, their reference count should never reach zero. There is also a dynamic table of user-defined *PyArray_Descr* objects that is also maintained. Once a data-type-descriptor object is "registered" it should never be deallocated either. The function *PyArray_DescrFromType* (…) can be used to retrieve a *PyArray_Descr* object from an enumerated type-number (either built-in or user-defined).

**PyArray_Descr**

The *PyArray_Descr* structure lies at the heart of the *PyArrayDescr_Type*. While it is described here for completeness, it should be considered internal to NumPy and manipulated via `PyArrayDescr_*` or `PyDataType*` functions and macros. The size of this structure is subject to change across versions of NumPy. To ensure compatibility:

- Never declare a non-pointer instance of the struct
- Never perform pointer arithmatic
- Never use `sizof(PyArray_Descr)`

It has the following structure:

```
typedef struct {
    PyObject_HEAD
    PyTypeObject *typeobj;
    char kind;
    char type;
    char byteorder;
    char flags;
    int type_num;
    int elsize;
    int alignment;
    PyArray_ArrayDescr *subarray;
    PyObject *fields;
    PyObject *names;
    PyArray_ArrFuncs *f;
    PyObject *metadata;
    NpyAuxData *c_metadata;
    npy_hash_t hash;
} PyArray_Descr;
```

PyTypeObject ***PyArray_Descr.typeobj**

Pointer to a typeobject that is the corresponding Python type for the elements of this array. For the builtin types, this points to the corresponding array scalar. For user-defined types, this should point to a user-defined typeobject. This typeobject can either inherit from array scalars or not. If it does not inherit from array scalars, then the *NPY_USE_GETITEM* and *NPY_USE_SETITEM* flags should be set in the `flags` member.

char **PyArray_Descr.kind**

A character code indicating the kind of array (using the array interface typestring notation). A 'b' represents Boolean, a 'i' represents signed integer, a 'u' represents unsigned integer, 'f' represents floating point, 'c' represents complex floating point, 'S' represents 8-bit zero-terminated bytes, 'U' represents 32-bit/character unicode string, and 'V' represents arbitrary.

char **PyArray_Descr.type**

A traditional character code indicating the data type.

char **PyArray_Descr.byteorder**

A character indicating the byte-order: '>' (big-endian), '<' (little- endian), '=' (native), '|' (irrelevant, ignore). All builtin data- types have byteorder '='.

char **PyArray_Descr.flags**

A data-type bit-flag that determines if the data-type exhibits object- array like behavior. Each bit in this member is a flag which are named as:

**NPY_ITEM_REFCOUNT**

Indicates that items of this data-type must be reference counted (using `Py_INCREF` and `Py_DECREF` ).

**NPY_ITEM_HASOBJECT**

Same as *NPY_ITEM_REFCOUNT*.

**NPY_LIST_PICKLE**
Indicates arrays of this data-type must be converted to a list before pickling.

**NPY_ITEM_IS_POINTER**
Indicates the item is a pointer to some other data-type

**NPY_NEEDS_INIT**
Indicates memory for this data-type must be initialized (set to 0) on creation.

**NPY_NEEDS_PYAPI**
Indicates this data-type requires the Python C-API during access (so don't give up the GIL if array access is going to be needed).

**NPY_USE_GETITEM**
On array access use the `f->getitem` function pointer instead of the standard conversion to an array scalar. Must use if you don't define an array scalar to go along with the data-type.

**NPY_USE_SETITEM**
When creating a 0-d array from an array scalar use `f->setitem` instead of the standard copy from an array scalar. Must use if you don't define an array scalar to go along with the data-type.

**NPY_FROM_FIELDS**
The bits that are inherited for the parent data-type if these bits are set in any field of the data-type. Currently ( *NPY_NEEDS_INIT* | *NPY_LIST_PICKLE* | *NPY_ITEM_REFCOUNT* | *NPY_NEEDS_PYAPI* ).

**NPY_OBJECT_DTYPE_FLAGS**
Bits set for the object data-type: ( *NPY_LIST_PICKLE* | *NPY_USE_GETITEM* | *NPY_ITEM_IS_POINTER* | `NPY_REFCOUNT` | *NPY_NEEDS_INIT* | *NPY_NEEDS_PYAPI*).

**PyDataType_FLAGCHK**(*PyArray_Descr *dtype*, int *flags*)
Return true if all the given flags are set for the data-type object.

**PyDataType_REFCHK**(*PyArray_Descr *dtype*)
Equivalent to *PyDataType_FLAGCHK* (*dtype*, *NPY_ITEM_REFCOUNT*).

int **PyArray_Descr.type_num**
A number that uniquely identifies the data type. For new data-types, this number is assigned when the data-type is registered.

int **PyArray_Descr.elsize**
For data types that are always the same size (such as long), this holds the size of the data type. For flexible data types where different arrays can have a different elementsize, this should be 0.

int **PyArray_Descr.alignment**
A number providing alignment information for this data type. Specifically, it shows how far from the start of a 2-element structure (whose first element is a `char` ), the compiler places an item of this type: `offsetof(struct {char c; type v;}, v)`

PyArray_ArrayDescr ***PyArray_Descr.subarray**
If this is non- `NULL`, then this data-type descriptor is a C-style contiguous array of another data-type descriptor. In other-words, each element that this descriptor describes is actually an array of some other base descriptor. This is most useful as the data-type descriptor for a field in another data-type descriptor. The fields member should be `NULL` if this is non- `NULL` (the fields member of the base descriptor can be non- `NULL` however). The `PyArray_ArrayDescr` structure is defined using

```
typedef struct {
    PyArray_Descr *base;
    PyObject *shape;
} PyArray_ArrayDescr;
```

The elements of this structure are:

*PyArray_Descr* ***PyArray_ArrayDescr.base**

The data-type-descriptor object of the base-type.

PyObject ***PyArray_ArrayDescr.shape**

The shape (always C-style contiguous) of the sub-array as a Python tuple.

PyObject ***PyArray_Descr.fields**

If this is non-NULL, then this data-type-descriptor has fields described by a Python dictionary whose keys are names (and also titles if given) and whose values are tuples that describe the fields. Recall that a data-type-descriptor always describes a fixed-length set of bytes. A field is a named sub-region of that total, fixed-length collection. A field is described by a tuple composed of another data- type-descriptor and a byte offset. Optionally, the tuple may contain a title which is normally a Python string. These tuples are placed in this dictionary keyed by name (and also title if given).

PyObject ***PyArray_Descr.names**

An ordered tuple of field names. It is NULL if no field is defined.

*PyArray_ArrFuncs* ***PyArray_Descr.f**

A pointer to a structure containing functions that the type needs to implement internal features. These functions are not the same thing as the universal functions (ufuncs) described later. Their signatures can vary arbitrarily.

PyObject ***PyArray_Descr.metadata**

Metadata about this dtype.

*NpyAuxData* ***PyArray_Descr.c_metadata**

Metadata specific to the C implementation of the particular dtype. Added for NumPy 1.7.0.

Npy_hash_t ***PyArray_Descr.hash**

Currently unused. Reserved for future use in caching hash values.

**PyArray_ArrFuncs**

Functions implementing internal features. Not all of these function pointers must be defined for a given type. The required members are `nonzero`, `copyswap`, `copyswapn`, `setitem`, `getitem`, and `cast`. These are assumed to be non- `NULL` and `NULL` entries will cause a program crash. The other functions may be `NULL` which will just mean reduced functionality for that data-type. (Also, the nonzero function will be filled in with a default function if it is `NULL` when you register a user-defined data-type).

```
typedef struct {
    PyArray_VectorUnaryFunc *cast[NPY_NTYPES];
    PyArray_GetItemFunc *getitem;
    PyArray_SetItemFunc *setitem;
    PyArray_CopySwapNFunc *copyswapn;
    PyArray_CopySwapFunc *copyswap;
    PyArray_CompareFunc *compare;
    PyArray_ArgFunc *argmax;
    PyArray_DotFunc *dotfunc;
    PyArray_ScanFunc *scanfunc;
    PyArray_FromStrFunc *fromstr;
    PyArray_NonzeroFunc *nonzero;
    PyArray_FillFunc *fill;
    PyArray_FillWithScalarFunc *fillwithscalar;
    PyArray_SortFunc *sort[NPY_NSORTS];
    PyArray_ArgSortFunc *argsort[NPY_NSORTS];
    PyObject *castdict;
    PyArray_ScalarKindFunc *scalarkind;
    int **cancastscalarkindto;
    int *cancastto;
    PyArray_FastClipFunc *fastclip;  /* deprecated */
    PyArray_FastPutmaskFunc *fastputmask;  /* deprecated */
```

(continues on next page)

(continued from previous page)

```
    PyArray_FastTakeFunc *fasttake;  /* deprecated */
    PyArray_ArgFunc *argmin;
} PyArray_ArrFuncs;
```

The concept of a behaved segment is used in the description of the function pointers. A behaved segment is one that is aligned and in native machine byte-order for the data-type. The `nonzero`, `copyswap`, `copyswapn`, `getitem`, and `setitem` functions can (and must) deal with mis-behaved arrays. The other functions require behaved memory segments.

void **cast** (void **from*, void **to*, *npy_intp* *n*, void **fromarr*, void **toarr*)

An array of function pointers to cast from the current type to all of the other builtin types. Each function casts a contiguous, aligned, and notswapped buffer pointed at by *from* to a contiguous, aligned, and notswapped buffer pointed at by *to* The number of items to cast is given by *n*, and the arguments *fromarr* and *toarr* are interpreted as PyArrayObjects for flexible arrays to get itemsize information.

PyObject ***getitem** (void **data*, void **arr*)

A pointer to a function that returns a standard Python object from a single element of the array object *arr* pointed to by *data*. This function must be able to deal with "misbehaved "(misaligned and/or swapped) arrays correctly.

int **setitem** (PyObject **item*, void **data*, void **arr*)

A pointer to a function that sets the Python object *item* into the array, *arr*, at the position pointed to by *data* . This function deals with "misbehaved" arrays. If successful, a zero is returned, otherwise, a negative one is returned (and a Python error set).

void **copyswapn** (void **dest*, *npy_intp* *dstride*, void **src*, *npy_intp* *sstride*, *npy_intp* *n*, int *swap*, void **arr*)

void **copyswap** (void **dest*, void **src*, int *swap*, void **arr*)

These members are both pointers to functions to copy data from *src* to *dest* and *swap* if indicated. The value of arr is only used for flexible ( `NPY_STRING`, `NPY_UNICODE`, and `NPY_VOID` ) arrays (and is obtained from `arr->descr->elsize` ). The second function copies a single value, while the first loops over n values with the provided strides. These functions can deal with misbehaved *src* data. If *src* is NULL then no copy is performed. If *swap* is 0, then no byteswapping occurs. It is assumed that *dest* and *src* do not overlap. If they overlap, then use `memmove` (…) first followed by `copyswap(n)` with NULL valued `src`.

int **compare** (const void* *d1*, const void* *d2*, void* *arr*)

A pointer to a function that compares two elements of the array, `arr`, pointed to by `d1` and `d2`. This function requires behaved (aligned and not swapped) arrays. The return value is 1 if * `d1` > * `d2`, 0 if * `d1` == * `d2`, and -1 if * `d1` < * `d2`. The array object `arr` is used to retrieve itemsize and field information for flexible arrays.

int **argmax** (void* *data*, *npy_intp* *n*, *npy_intp** *max_ind*, void* *arr*)

A pointer to a function that retrieves the index of the largest of `n` elements in `arr` beginning at the element pointed to by `data`. This function requires that the memory segment be contiguous and behaved. The return value is always 0. The index of the largest element is returned in `max_ind`.

void **dotfunc** (void* *ip1*, *npy_intp* *is1*, void* *ip2*, *npy_intp* *is2*, void* *op*, *npy_intp* *n*, void* *arr*)

A pointer to a function that multiplies two `n` -length sequences together, adds them, and places the result in element pointed to by `op` of `arr`. The start of the two sequences are pointed to by `ip1` and `ip2`. To get to the next element in each sequence requires a jump of `is1` and `is2` *bytes*, respectively. This function requires behaved (though not necessarily contiguous) memory.

int **scanfunc** (FILE* *fd*, void* *ip*, void* *arr*)

A pointer to a function that scans (scanf style) one element of the corresponding type from the file descriptor `fd` into the array memory pointed to by `ip`. The array is assumed to be behaved. The last argument `arr` is the array to be scanned into. Returns number of receiving arguments successfully assigned (which may be zero in case a matching failure occurred before the first receiving argument was assigned), or EOF if input failure occurs before the first receiving argument was assigned. This function should be called without holding the Python GIL, and has to grab it for error reporting.

int **fromstr** (char* *str*, void* *ip*, char** *endptr*, void* *arr*)

A pointer to a function that converts the string pointed to by `str` to one element of the corresponding type and places it in the memory location pointed to by `ip`. After the conversion is completed, `*endptr` points to the rest of the string. The last argument `arr` is the array into which ip points (needed for variable-size data- types). Returns 0 on success or -1 on failure. Requires a behaved array. This function should be called without holding the Python GIL, and has to grab it for error reporting.

Bool **nonzero** (void* *data*, void* *arr*)

A pointer to a function that returns TRUE if the item of `arr` pointed to by `data` is nonzero. This function can deal with misbehaved arrays.

void **fill** (void* *data*, *npy_intp* *length*, void* *arr*)

A pointer to a function that fills a contiguous array of given length with data. The first two elements of the array must already be filled- in. From these two values, a delta will be computed and the values from item 3 to the end will be computed by repeatedly adding this computed delta. The data buffer must be well-behaved.

void **fillwithscalar** (void* *buffer*, *npy_intp* *length*, void* *value*, void* *arr*)

A pointer to a function that fills a contiguous `buffer` of the given `length` with a single scalar `value` whose address is given. The final argument is the array which is needed to get the itemsize for variable-length arrays.

int **sort** (void* *start*, *npy_intp* *length*, void* *arr*)

An array of function pointers to a particular sorting algorithms. A particular sorting algorithm is obtained using a key (so far `NPY_QUICKSORT`, `NPY_HEAPSORT`, and `NPY_MERGESORT` are defined). These sorts are done in-place assuming contiguous and aligned data.

int **argsort** (void* *start*, *npy_intp** *result*, *npy_intp* *length*, void **arr*)

An array of function pointers to sorting algorithms for this data type. The same sorting algorithms as for sort are available. The indices producing the sort are returned in `result` (which must be initialized with indices 0 to `length-1` inclusive).

PyObject ***castdict**

Either `NULL` or a dictionary containing low-level casting functions for user- defined data-types. Each function is wrapped in a `PyCObject *` and keyed by the data-type number.

*NPY_SCALARKIND* **scalarkind** (*PyArrayObject** *arr*)

A function to determine how scalars of this type should be interpreted. The argument is `NULL` or a 0-dimensional array containing the data (if that is needed to determine the kind of scalar). The return value must be of type `NPY_SCALARKIND`.

int ****cancastscalarkindto**

Either `NULL` or an array of `NPY_NSCALARKINDS` pointers. These pointers should each be either `NULL` or a pointer to an array of integers (terminated by `NPY_NOTYPE`) indicating data-types that a scalar of this data-type of the specified kind can be cast to safely (this usually means without losing precision).

int ***cancastto**

Either `NULL` or an array of integers (terminated by `NPY_NOTYPE` ) indicated data-types that this data-type can be cast to safely (this usually means without losing precision).

void **fastclip** (void **in*, *npy_intp* *n_in*, void **min*, void **max*, void **out*)

Deprecated since version 1.17: The use of this function will give a deprecation warning when `np.clip`. Instead of this function, the datatype must instead use `PyUFunc_RegisterLoopForDescr` to attach a custom loop to `np.core.umath.clip`, `np.minimum`, and `np.maximum`.

Deprecated since version 1.19: Setting this function is deprecated and should always be `NULL`, if set, it will be ignored.

A function that reads `n_in` items from `in`, and writes to `out` the read value if it is within the limits pointed to by `min` and `max`, or the corresponding limit if outside. The memory segments must be contiguous and behaved, and either `min` or `max` may be `NULL`, but not both.

void **`fastputmask`**(void **in*, void **mask*, *npy_intp* *n_in*, void **values*, *npy_intp* *nv*)

Deprecated since version 1.19: Setting this function is deprecated and should always be `NULL`, if set, it will be ignored.

A function that takes a pointer `in` to an array of `n_in` items, a pointer `mask` to an array of `n_in` boolean values, and a pointer `vals` to an array of `nv` items. Items from `vals` are copied into `in` wherever the value in `mask` is non-zero, tiling `vals` as needed if `nv < n_in`. All arrays must be contiguous and behaved.

void **`fasttake`**(void **dest*, void **src*, *npy_intp* **indarray*, *npy_intp* *nindarray*, *npy_intp* *n_outer*, *npy_intp* *m_middle*, *npy_intp* *nelem*, *NPY_CLIPMODE* *clipmode*)

Deprecated since version 1.19: Setting this function is deprecated and should always be `NULL`, if set, it will be ignored.

A function that takes a pointer `src` to a C contiguous, behaved segment, interpreted as a 3-dimensional array of shape `(n_outer, nindarray, nelem)`, a pointer `indarray` to a contiguous, behaved segment of `m_middle` integer indices, and a pointer `dest` to a C contiguous, behaved segment, interpreted as a 3-dimensional array of shape `(n_outer, m_middle, nelem)`. The indices in `indarray` are used to index `src` along the second dimension, and copy the corresponding chunks of `nelem` items into `dest`. `clipmode` (which can take on the values *`NPY_RAISE`*, *`NPY_WRAP`* or *`NPY_CLIP`*) determines how will indices smaller than 0 or larger than `nindarray` will be handled.

int **`argmin`**(void* *data*, *npy_intp* *n*, *npy_intp** *min_ind*, void* *arr*)

A pointer to a function that retrieves the index of the smallest of `n` elements in `arr` beginning at the element pointed to by `data`. This function requires that the memory segment be contiguous and behaved. The return value is always 0. The index of the smallest element is returned in `min_ind`.

The *`PyArray_Type`* typeobject implements many of the features of `Python objects` including the `tp_as_number`, `tp_as_sequence`, `tp_as_mapping`, and `tp_as_buffer` interfaces. The `rich comparison`) is also used along with new-style attribute lookup for member (`tp_members`) and properties (`tp_getset`). The *`PyArray_Type`* can also be sub-typed.

---

**Tip:** The `tp_as_number` methods use a generic approach to call whatever function has been registered for handling the operation. When the `_multiarray_umath module` is imported, it sets the numeric operations for all arrays to the corresponding ufuncs. This choice can be changed with *`PyUFunc_ReplaceLoopBySignature`* The `tp_str` and `tp_repr` methods can also be altered using *`PyArray_SetStringFunction`*.

---

### PyUFunc_Type and PyUFuncObject

**`PyUFunc_Type`**

The ufunc object is implemented by creation of the *`PyUFunc_Type`*. It is a very simple type that implements only basic getattribute behavior, printing behavior, and has call behavior which allows these objects to act like functions. The basic idea behind the ufunc is to hold a reference to fast 1-dimensional (vector) loops for each data type that supports the operation. These one-dimensional loops all have the same signature and are the key to creating a new ufunc. They are called by the generic looping code as appropriate to implement the N-dimensional function. There are also some generic 1-d loops defined for floating and complexfloating arrays that allow you to define a ufunc using a single scalar function (*e.g.* atanh).

**`PyUFuncObject`**

The core of the ufunc is the *`PyUFuncObject`* which contains all the information needed to call the underlying C-code loops that perform the actual work. While it is described here for completeness, it should be considered internal to NumPy and manipulated via `PyUFunc_*` functions. The size of this structure is subject to change across versions of NumPy. To ensure compatibility:

- Never declare a non-pointer instance of the struct
- Never perform pointer arithmetic
- Never use `sizeof(PyUFuncObject)`

It has the following structure:

```
typedef struct {
    PyObject_HEAD
    int nin;
    int nout;
    int nargs;
    int identity;
    PyUFuncGenericFunction *functions;
    void **data;
    int ntypes;
    int reserved1;
    const char *name;
    char *types;
    const char *doc;
    void *ptr;
    PyObject *obj;
    PyObject *userloops;
    int core_enabled;
    int core_num_dim_ix;
    int *core_num_dims;
    int *core_dim_ixs;
    int *core_offsets;
    char *core_signature;
    PyUFunc_TypeResolutionFunc *type_resolver;
    PyUFunc_LegacyInnerLoopSelectionFunc *legacy_inner_loop_selector;
    PyUFunc_MaskedInnerLoopSelectionFunc *masked_inner_loop_selector;
    npy_uint32 *op_flags;
    npy_uint32 *iter_flags;
    /* new in API version 0x0000000D */
    npy_intp *core_dim_sizes;
    npy_intp *core_dim_flags;

} PyUFuncObject;
```

int **`PyUFuncObject.nin`**

The number of input arguments.

int **`PyUFuncObject.nout`**

The number of output arguments.

int **`PyUFuncObject.nargs`**

The total number of arguments (*nin* + *nout*). This must be less than `NPY_MAXARGS`.

int **`PyUFuncObject.identity`**

Either `PyUFunc_One`, `PyUFunc_Zero`, `PyUFunc_None` or `PyUFunc_AllOnes` to indicate the identity for this operation. It is only used for a reduce-like call on an empty array.

void **`PyUFuncObject.functions`** (char** *args*, *npy_intp** *dims*, *npy_intp** *steps*, void* *extradata*)

An array of function pointers — one for each data type supported by the ufunc. This is the vector loop that is called to implement the underlying function *dims* [0] times. The first argument, *args*, is an array of *nargs* pointers to behaved memory. Pointers to the data for the input arguments are first, followed by the pointers to the data for the output arguments. How many bytes must be skipped to get to the next element in the sequence is specified by the corresponding entry in the *steps* array. The last argument allows the loop to receive extra information. This is commonly used so that a single, generic vector loop can be used for multiple functions. In this case, the actual scalar function to call is passed in as *extradata*. The size of this function pointer array is ntypes.

void ****`PyUFuncObject.data`**

Extra data to be passed to the 1-d vector loops or `NULL` if no extra-data is needed. This C-array must be the same size ( *i.e.* ntypes) as the functions array. `NULL` is used if extra_data is not needed. Several C-API calls for UFuncs are just 1-d vector loops that make use of this extra data to receive a pointer to the actual function to call.

int **`PyUFuncObject.ntypes`**

The number of supported data types for the ufunc. This number specifies how many different 1-d loops (of the builtin data types) are available.

int **`PyUFuncObject.reserved1`**

Unused.

char ***`PyUFuncObject.name`**

A string name for the ufunc. This is used dynamically to build the __doc__ attribute of ufuncs.

char ***`PyUFuncObject.types`**

An array of $nargs \times ntypes$ 8-bit type_numbers which contains the type signature for the function for each of the supported (builtin) data types. For each of the *ntypes* functions, the corresponding set of type numbers in this array shows how the *args* argument should be interpreted in the 1-d vector loop. These type numbers do not have to be the same type and mixed-type ufuncs are supported.

char ***`PyUFuncObject.doc`**

Documentation for the ufunc. Should not contain the function signature as this is generated dynamically when __doc__ is retrieved.

void ***`PyUFuncObject.ptr`**

Any dynamically allocated memory. Currently, this is used for dynamic ufuncs created from a python function to store room for the types, data, and name members.

PyObject ***`PyUFuncObject.obj`**

For ufuncs dynamically created from python functions, this member holds a reference to the underlying Python function.

PyObject ***`PyUFuncObject.userloops`**

A dictionary of user-defined 1-d vector loops (stored as CObject ptrs) for user-defined types. A loop may be registered by the user for any user-defined type. It is retrieved by type number. User defined type numbers are always larger than *`NPY_USERDEF`*.

int **`PyUFuncObject.core_enabled`**

0 for scalar ufuncs; 1 for generalized ufuncs

int **`PyUFuncObject.core_num_dim_ix`**

Number of distinct core dimension names in the signature

int ***`PyUFuncObject.core_num_dims`**

Number of core dimensions of each argument

int ***`PyUFuncObject.core_dim_ixs`**

Dimension indices in a flattened form; indices of argument `k` are stored in `core_dim_ixs[core_offsets[k] : core_offsets[k] + core_numdims[k]]`

int ***`PyUFuncObject.core_offsets`**

Position of 1st core dimension of each argument in `core_dim_ixs`, equivalent to cumsum(`core_num_dims`)

char ***`PyUFuncObject.core_signature`**
Core signature string

PyUFunc_TypeResolutionFunc ***`PyUFuncObject.type_resolver`**
A function which resolves the types and fills an array with the dtypes for the inputs and outputs

PyUFunc_LegacyInnerLoopSelectionFunc ***`PyUFuncObject.legacy_inner_loop_selector`**
A function which returns an inner loop. The `legacy` in the name arises because for NumPy 1.6 a better variant had been planned. This variant has not yet come about.

void ***`PyUFuncObject.reserved2`**
For a possible future loop selector with a different signature.

PyUFunc_MaskedInnerLoopSelectionFunc ***`PyUFuncObject.masked_inner_loop_selector`**
Function which returns a masked inner loop for the ufunc

*npy_uint32* **`PyUFuncObject.op_flags`**
Override the default operand flags for each ufunc operand.

*npy_uint32* **`PyUFuncObject.iter_flags`**
Override the default nditer flags for the ufunc.

Added in API version 0x0000000D

*npy_intp* ***`PyUFuncObject.core_dim_sizes`**
For each distinct core dimension, the possible *frozen* size if `UFUNC_CORE_DIM_SIZE_INFERRED` is 0

*npy_uint32* ***`PyUFuncObject.core_dim_flags`**
For each distinct core dimension, a set of `UFUNC_CORE_DIM*` flags

- `UFUNC_CORE_DIM_CAN_IGNORE` if the dim name ends in `?`
- `UFUNC_CORE_DIM_SIZE_INFERRED` if the dim size will be determined from the operands and not from a *frozen* signature

### PyArrayIter_Type and PyArrayIterObject

**`PyArrayIter_Type`**
This is an iterator object that makes it easy to loop over an N-dimensional array. It is the object returned from the flat attribute of an ndarray. It is also used extensively throughout the implementation internals to loop over an N-dimensional array. The tp_as_mapping interface is implemented so that the iterator object can be indexed (using 1-d indexing), and a few methods are implemented through the tp_methods table. This object implements the next method and can be used anywhere an iterator can be used in Python.

**`PyArrayIterObject`**
The C-structure corresponding to an object of `PyArrayIter_Type` is the `PyArrayIterObject`. The `PyArrayIterObject` is used to keep track of a pointer into an N-dimensional array. It contains associated information used to quickly march through the array. The pointer can be adjusted in three basic ways: 1) advance to the "next" position in the array in a C-style contiguous fashion, 2) advance to an arbitrary N-dimensional co-ordinate in the array, and 3) advance to an arbitrary one-dimensional index into the array. The members of the `PyArrayIterObject` structure are used in these calculations. Iterator objects keep their own dimension and strides information about an array. This can be adjusted as needed for "broadcasting," or to loop over only specific dimensions.

```
typedef struct {
    PyObject_HEAD
    int   nd_m1;
    npy_intp  index;
    npy_intp  size;
```

(continues on next page)

(continued from previous page)

```
    npy_intp  coordinates[NPY_MAXDIMS];
    npy_intp  dims_m1[NPY_MAXDIMS];
    npy_intp  strides[NPY_MAXDIMS];
    npy_intp  backstrides[NPY_MAXDIMS];
    npy_intp  factors[NPY_MAXDIMS];
    PyArrayObject *ao;
    char  *dataptr;
    Bool  contiguous;
} PyArrayIterObject;
```

int **`PyArrayIterObject.nd_m1`**

$N-1$ where $N$ is the number of dimensions in the underlying array.

*npy_intp* **`PyArrayIterObject.index`**

The current 1-d index into the array.

*npy_intp* **`PyArrayIterObject.size`**

The total size of the underlying array.

*npy_intp* ***`PyArrayIterObject.coordinates`**

An $N$ -dimensional index into the array.

*npy_intp* ***`PyArrayIterObject.dims_m1`**

The size of the array minus 1 in each dimension.

*npy_intp* ***`PyArrayIterObject.strides`**

The strides of the array. How many bytes needed to jump to the next element in each dimension.

*npy_intp* ***`PyArrayIterObject.backstrides`**

How many bytes needed to jump from the end of a dimension back to its beginning. Note that `backstrides[k] == strides[k] * dims_m1[k]`, but it is stored here as an optimization.

*npy_intp* ***`PyArrayIterObject.factors`**

This array is used in computing an N-d index from a 1-d index. It contains needed products of the dimensions.

*PyArrayObject* ***`PyArrayIterObject.ao`**

A pointer to the underlying ndarray this iterator was created to represent.

char ***`PyArrayIterObject.dataptr`**

This member points to an element in the ndarray indicated by the index.

Bool **`PyArrayIterObject.contiguous`**

This flag is true if the underlying array is `NPY_ARRAY_C_CONTIGUOUS`. It is used to simplify calculations when possible.

How to use an array iterator on a C-level is explained more fully in later sections. Typically, you do not need to concern yourself with the internal structure of the iterator object, and merely interact with it through the use of the macros `PyArray_ITER_NEXT` (it), `PyArray_ITER_GOTO` (it, dest), or `PyArray_ITER_GOTO1D` (it, index). All of these macros require the argument *it* to be a `PyArrayIterObject *`.

### PyArrayMultiIter_Type and PyArrayMultiIterObject

**`PyArrayMultiIter_Type`**

This type provides an iterator that encapsulates the concept of broadcasting. It allows $N$ arrays to be broadcast together so that the loop progresses in C-style contiguous fashion over the broadcasted array. The corresponding C-structure is the `PyArrayMultiIterObject` whose memory layout must begin any object, *obj*, passed in to the `PyArray_Broadcast` (obj) function. Broadcasting is performed by adjusting array iterators so that each iterator represents the broadcasted shape and size, but has its strides adjusted so that the correct element from the array is used at each iteration.

**`PyArrayMultiIterObject`**

```
typedef struct {
    PyObject_HEAD
    int numiter;
    npy_intp size;
    npy_intp index;
    int nd;
    npy_intp dimensions[NPY_MAXDIMS];
    PyArrayIterObject *iters[NPY_MAXDIMS];
} PyArrayMultiIterObject;
```

int **`PyArrayMultiIterObject.numiter`**

The number of arrays that need to be broadcast to the same shape.

*npy_intp* **`PyArrayMultiIterObject.size`**

The total broadcasted size.

*npy_intp* **`PyArrayMultiIterObject.index`**

The current (1-d) index into the broadcasted result.

int **`PyArrayMultiIterObject.nd`**

The number of dimensions in the broadcasted result.

*npy_intp* ***`PyArrayMultiIterObject.dimensions`**

The shape of the broadcasted result (only `nd` slots are used).

*PyArrayIterObject* ****`PyArrayMultiIterObject.iters`**

An array of iterator objects that holds the iterators for the arrays to be broadcast together. On return, the iterators are adjusted for broadcasting.

### PyArrayNeighborhoodIter_Type and PyArrayNeighborhoodIterObject

**`PyArrayNeighborhoodIter_Type`**

This is an iterator object that makes it easy to loop over an N-dimensional neighborhood.

**`PyArrayNeighborhoodIterObject`**

The C-structure corresponding to an object of `PyArrayNeighborhoodIter_Type` is the `PyArrayNeighborhoodIterObject`.

```
typedef struct {
    PyObject_HEAD
    int nd_m1;
    npy_intp index, size;
    npy_intp coordinates[NPY_MAXDIMS]
    npy_intp dims_m1[NPY_MAXDIMS];
```

(continues on next page)

(continued from previous page)

```
    npy_intp strides[NPY_MAXDIMS];
    npy_intp backstrides[NPY_MAXDIMS];
    npy_intp factors[NPY_MAXDIMS];
    PyArrayObject *ao;
    char *dataptr;
    npy_bool contiguous;
    npy_intp bounds[NPY_MAXDIMS][2];
    npy_intp limits[NPY_MAXDIMS][2];
    npy_intp limits_sizes[NPY_MAXDIMS];
    npy_iter_get_dataptr_t translate;
    npy_intp nd;
    npy_intp dimensions[NPY_MAXDIMS];
    PyArrayIterObject* _internal_iter;
    char* constant;
    int mode;
} PyArrayNeighborhoodIterObject;
```

### PyArrayFlags_Type and PyArrayFlagsObject

**`PyArrayFlags_Type`**

When the flags attribute is retrieved from Python, a special builtin object of this type is constructed. This special type makes it easier to work with the different flags by accessing them as attributes or by accessing them as if the object were a dictionary with the flag names as entries.

**`PyArrayFlagsObject`**

```
typedef struct PyArrayFlagsObject {
        PyObject_HEAD
        PyObject *arr;
        int flags;
} PyArrayFlagsObject;
```

### ScalarArrayTypes

There is a Python type for each of the different built-in data types that can be present in the array Most of these are simple wrappers around the corresponding data type in C. The C-names for these types are `Py{TYPE}ArrType_Type` where `{TYPE}` can be

> **Bool**, **Byte**, **Short**, **Int**, **Long**, **LongLong**, **UByte**, **UShort**, **UInt**, **ULong**, **ULongLong**, **Half**, **Float**, **Double**, **LongDouble**, **CFloat**, **CDouble**, **CLongDouble**, **String**, **Unicode**, **Void**, and **Object**.

These type names are part of the C-API and can therefore be created in extension C-code. There is also a `PyIntpArrType_Type` and a `PyUIntpArrType_Type` that are simple substitutes for one of the integer types that can hold a pointer on the platform. The structure of these scalar objects is not exposed to C-code. The function *`PyArray_ScalarAsCtype`* (..) can be used to extract the C-type value from the array scalar and the function *`PyArray_Scalar`* (…) can be used to construct an array scalar from a C-value.

## 8.1.2 Other C-Structures

A few new C-structures were found to be useful in the development of NumPy. These C-structures are used in at least one C-API call and are therefore documented here. The main reason these structures were defined is to make it easy to use the Python ParseTuple C-API to convert from Python objects to a useful C-Object.

### PyArray_Dims

**`PyArray_Dims`**

This structure is very useful when shape and/or strides information is supposed to be interpreted. The structure is:

```
typedef struct {
    npy_intp *ptr;
    int len;
} PyArray_Dims;
```

The members of this structure are

*npy_intp* ***`PyArray_Dims.ptr`***

A pointer to a list of (*`npy_intp`*) integers which usually represent array shape or array strides.

int **`PyArray_Dims.len`**

The length of the list of integers. It is assumed safe to access *ptr* [0] to *ptr* [len-1].

### PyArray_Chunk

**`PyArray_Chunk`**

This is equivalent to the buffer object structure in Python up to the ptr member. On 32-bit platforms (*i.e.* if *`NPY_SIZEOF_INT`* == `NPY_SIZEOF_INTP`), the len member also matches an equivalent member of the buffer object. It is useful to represent a generic single-segment chunk of memory.

```
typedef struct {
    PyObject_HEAD
    PyObject *base;
    void *ptr;
    npy_intp len;
    int flags;
} PyArray_Chunk;
```

The members are

PyObject ***`PyArray_Chunk.base`***

The Python object this chunk of memory comes from. Needed so that memory can be accounted for properly.

void ***`PyArray_Chunk.ptr`***

A pointer to the start of the single-segment chunk of memory.

*npy_intp* **`PyArray_Chunk.len`**

The length of the segment in bytes.

int **`PyArray_Chunk.flags`**

Any data flags (*e.g.* *`NPY_ARRAY_WRITEABLE`* ) that should be used to interpret the memory.

### PyArrayInterface

**See also:**

*The Array Interface*

**`PyArrayInterface`**

The `PyArrayInterface` structure is defined so that NumPy and other extension modules can use the rapid array interface protocol. The `__array_struct__` method of an object that supports the rapid array interface protocol should return a `PyCObject` that contains a pointer to a `PyArrayInterface` structure with the relevant details of the array. After the new array is created, the attribute should be `DECREF`'d which will free the `PyArrayInterface` structure. Remember to `INCREF` the object (whose `__array_struct__` attribute was retrieved) and point the base member of the new `PyArrayObject` to this same object. In this way the memory for the array will be managed correctly.

```
typedef struct {
    int two;
    int nd;
    char typekind;
    int itemsize;
    int flags;
    npy_intp *shape;
    npy_intp *strides;
    void *data;
    PyObject *descr;
} PyArrayInterface;
```

int **`PyArrayInterface.two`**

the integer 2 as a sanity check.

int **`PyArrayInterface.nd`**

the number of dimensions in the array.

char **`PyArrayInterface.typekind`**

A character indicating what kind of array is present according to the typestring convention with 't' -> bitfield, 'b' -> Boolean, 'i' -> signed integer, 'u' -> unsigned integer, 'f' -> floating point, 'c' -> complex floating point, 'O' -> object, 'S' -> (byte-)string, 'U' -> unicode, 'V' -> void.

int **`PyArrayInterface.itemsize`**

The number of bytes each item in the array requires.

int **`PyArrayInterface.flags`**

Any of the bits *`NPY_ARRAY_C_CONTIGUOUS`* (1), *`NPY_ARRAY_F_CONTIGUOUS`* (2), *`NPY_ARRAY_ALIGNED`* (0x100), *`NPY_ARRAY_NOTSWAPPED`* (0x200), or *`NPY_ARRAY_WRITEABLE`* (0x400) to indicate something about the data. The *`NPY_ARRAY_ALIGNED`*, *`NPY_ARRAY_C_CONTIGUOUS`*, and *`NPY_ARRAY_F_CONTIGUOUS`* flags can actually be determined from the other parameters. The flag `NPY_ARR_HAS_DESCR` (0x800) can also be set to indicate to objects consuming the version 3 array interface that the descr member of the structure is present (it will be ignored by objects consuming version 2 of the array interface).

*npy_intp* ***`PyArrayInterface.shape`**

An array containing the size of the array in each dimension.

*npy_intp* ***`PyArrayInterface.strides`**

An array containing the number of bytes to jump to get to the next element in each dimension.

void ***`PyArrayInterface.data`**

A pointer *to* the first element of the array.

PyObject ***PyArrayInterface.descr**

A Python object describing the data-type in more detail (same as the *descr* key in `__array_interface__`). This can be `NULL` if *typekind* and *itemsize* provide enough information. This field is also ignored unless `ARR_HAS_DESCR` flag is on in *flags*.

### Internally used structures

Internally, the code uses some additional Python objects primarily for memory management. These types are not accessible directly from Python, and are not exposed to the C-API. They are included here only for completeness and assistance in understanding the code.

**PyUFuncLoopObject**

A loose wrapper for a C-structure that contains the information needed for looping. This is useful if you are trying to understand the ufunc looping code. The *PyUFuncLoopObject* is the associated C-structure. It is defined in the `ufuncobject.h` header.

**PyUFuncReduceObject**

A loose wrapper for the C-structure that contains the information needed for reduce-like methods of ufuncs. This is useful if you are trying to understand the reduce, accumulate, and reduce-at code. The *PyUFuncReduceObject* is the associated C-structure. It is defined in the `ufuncobject.h` header.

**PyUFunc_Loop1d**

A simple linked-list of C-structures containing the information needed to define a 1-d loop for a ufunc for every defined signature of a user-defined data-type.

**PyArrayMapIter_Type**

Advanced indexing is handled with this Python type. It is simply a loose wrapper around the C-structure containing the variables needed for advanced array indexing. The associated C-structure, `PyArrayMapIterObject`, is useful if you are trying to understand the advanced-index mapping code. It is defined in the `arrayobject.h` header. This type is not exposed to Python and could be replaced with a C-structure. As a Python type it takes advantage of reference- counted memory management.

## 8.2 System configuration

When NumPy is built, information about system configuration is recorded, and is made available for extension modules using NumPy's C API. These are mostly defined in `numpyconfig.h` (included in `ndarrayobject.h`). The public symbols are prefixed by `NPY_*`. NumPy also offers some functions for querying information about the platform in use.

For private use, NumPy also constructs a `config.h` in the NumPy include directory, which is not exported by NumPy (that is a python extension which use the numpy C API will not see those symbols), to avoid namespace pollution.

### 8.2.1 Data type sizes

The `NPY_SIZEOF_{CTYPE}` constants are defined so that sizeof information is available to the pre-processor.

**NPY_SIZEOF_SHORT**

sizeof(short)

**NPY_SIZEOF_INT**

sizeof(int)

**NPY_SIZEOF_LONG**

sizeof(long)

**`NPY_SIZEOF_LONGLONG`**
sizeof(longlong) where longlong is defined appropriately on the platform.

**`NPY_SIZEOF_PY_LONG_LONG`**

**`NPY_SIZEOF_FLOAT`**
sizeof(float)

**`NPY_SIZEOF_DOUBLE`**
sizeof(double)

**`NPY_SIZEOF_LONG_DOUBLE`**
sizeof(longdouble) (A macro defines **NPY_SIZEOF_LONGDOUBLE** as well.)

**`NPY_SIZEOF_PY_INTPTR_T`**
Size of a pointer on this platform (sizeof(void *)) (A macro defines NPY_SIZEOF_INTP as well.)

### 8.2.2 Platform information

**`NPY_CPU_X86`**

**`NPY_CPU_AMD64`**

**`NPY_CPU_IA64`**

**`NPY_CPU_PPC`**

**`NPY_CPU_PPC64`**

**`NPY_CPU_SPARC`**

**`NPY_CPU_SPARC64`**

**`NPY_CPU_S390`**

**`NPY_CPU_PARISC`**
New in version 1.3.0.

CPU architecture of the platform; only one of the above is defined.

Defined in `numpy/npy_cpu.h`

**`NPY_LITTLE_ENDIAN`**

**`NPY_BIG_ENDIAN`**

**`NPY_BYTE_ORDER`**
New in version 1.3.0.

Portable alternatives to the `endian.h` macros of GNU Libc. If big endian, *`NPY_BYTE_ORDER`* == *`NPY_BIG_ENDIAN`*, and similarly for little endian architectures.

Defined in `numpy/npy_endian.h`.

**`PyArray_GetEndianness`**`()`
New in version 1.3.0.

Returns the endianness of the current platform. One of `NPY_CPU_BIG`, `NPY_CPU_LITTLE`, or `NPY_CPU_UNKNOWN_ENDIAN`.

### 8.2.3 Compiler directives

**`NPY_LIKELY`**

**`NPY_UNLIKELY`**

**`NPY_UNUSED`**

### 8.2.4 Interrupt Handling

**`NPY_INTERRUPT_H`**

**`NPY_SIGSETJMP`**

**`NPY_SIGLONGJMP`**

**`NPY_SIGJMP_BUF`**

**`NPY_SIGINT_ON`**

**`NPY_SIGINT_OFF`**

## 8.3 Data Type API

The standard array can have 24 different data types (and has some support for adding your own types). These data types all have an enumerated type, an enumerated type-character, and a corresponding array scalar Python type object (placed in a hierarchy). There are also standard C typedefs to make it easier to manipulate elements of the given data type. For the numeric types, there are also bit-width equivalent C typedefs and named typenumbers that make it easier to select the precision desired.

> **Warning:** The names for the types in c code follows c naming conventions more closely. The Python names for these types follow Python conventions. Thus, *`NPY_FLOAT`* picks up a 32-bit float in C, but `numpy.float_` in Python corresponds to a 64-bit double. The bit-width names can be used in both Python and C for clarity.

### 8.3.1 Enumerated Types

**`NPY_TYPES`**

There is a list of enumerated types defined providing the basic 24 data types plus some useful generic names. Whenever the code requires a type number, one of these enumerated types is requested. The types are all called `NPY_{NAME}`:

**`NPY_BOOL`**
The enumeration value for the boolean type, stored as one byte. It may only be set to the values 0 and 1.

**`NPY_BYTE`**

**`NPY_INT8`**
The enumeration value for an 8-bit/1-byte signed integer.

**`NPY_SHORT`**

**`NPY_INT16`**
The enumeration value for a 16-bit/2-byte signed integer.

**`NPY_INT`**

**NPY_INT32**
The enumeration value for a 32-bit/4-byte signed integer.

**NPY_LONG**
Equivalent to either NPY_INT or NPY_LONGLONG, depending on the platform.

**NPY_LONGLONG**

**NPY_INT64**
The enumeration value for a 64-bit/8-byte signed integer.

**NPY_UBYTE**

**NPY_UINT8**
The enumeration value for an 8-bit/1-byte unsigned integer.

**NPY_USHORT**

**NPY_UINT16**
The enumeration value for a 16-bit/2-byte unsigned integer.

**NPY_UINT**

**NPY_UINT32**
The enumeration value for a 32-bit/4-byte unsigned integer.

**NPY_ULONG**
Equivalent to either NPY_UINT or NPY_ULONGLONG, depending on the platform.

**NPY_ULONGLONG**

**NPY_UINT64**
The enumeration value for a 64-bit/8-byte unsigned integer.

**NPY_HALF**

**NPY_FLOAT16**
The enumeration value for a 16-bit/2-byte IEEE 754-2008 compatible floating point type.

**NPY_FLOAT**

**NPY_FLOAT32**
The enumeration value for a 32-bit/4-byte IEEE 754 compatible floating point type.

**NPY_DOUBLE**

**NPY_FLOAT64**
The enumeration value for a 64-bit/8-byte IEEE 754 compatible floating point type.

**NPY_LONGDOUBLE**
The enumeration value for a platform-specific floating point type which is at least as large as NPY_DOUBLE, but larger on many platforms.

**NPY_CFLOAT**

**NPY_COMPLEX64**
The enumeration value for a 64-bit/8-byte complex type made up of two NPY_FLOAT values.

**NPY_CDOUBLE**

**NPY_COMPLEX128**
The enumeration value for a 128-bit/16-byte complex type made up of two NPY_DOUBLE values.

**NPY_CLONGDOUBLE**
The enumeration value for a platform-specific complex floating point type which is made up of two NPY_LONGDOUBLE values.

**NPY_DATETIME**
The enumeration value for a data type which holds dates or datetimes with a precision based on selectable date or time units.

**NPY_TIMEDELTA**
The enumeration value for a data type which holds lengths of times in integers of selectable date or time units.

**NPY_STRING**
The enumeration value for ASCII strings of a selectable size. The strings have a fixed maximum size within a given array.

**NPY_UNICODE**
The enumeration value for UCS4 strings of a selectable size. The strings have a fixed maximum size within a given array.

**NPY_OBJECT**
The enumeration value for references to arbitrary Python objects.

**NPY_VOID**
Primarily used to hold struct dtypes, but can contain arbitrary binary data.

Some useful aliases of the above types are

**NPY_INTP**
The enumeration value for a signed integer type which is the same size as a (void *) pointer. This is the type used by all arrays of indices.

**NPY_UINTP**
The enumeration value for an unsigned integer type which is the same size as a (void *) pointer.

**NPY_MASK**
The enumeration value of the type used for masks, such as with the *`NPY_ITER_ARRAYMASK`* iterator flag. This is equivalent to *`NPY_UINT8`*.

**NPY_DEFAULT_TYPE**
The default type to use when no dtype is explicitly specified, for example when calling np.zero(shape). This is equivalent to *`NPY_DOUBLE`*.

Other useful related constants are

**NPY_NTYPES**
The total number of built-in NumPy types. The enumeration covers the range from 0 to NPY_NTYPES-1.

**NPY_NOTYPE**
A signal value guaranteed not to be a valid type enumeration number.

**NPY_USERDEF**
The start of type numbers used for Custom Data types.

The various character codes indicating certain types are also part of an enumerated list. References to type characters (should they be needed at all) should always use these enumerations. The form of them is `NPY_{NAME}LTR` where `{NAME}` can be

**BOOL**, **BYTE**, **UBYTE**, **SHORT**, **USHORT**, **INT**, **UINT**, **LONG**, **ULONG**, **LONGLONG**, **ULONGLONG**, **HALF**, **FLOAT**, **DOUBLE**, **LONGDOUBLE**, **CFLOAT**, **CDOUBLE**, **CLONGDOUBLE**, **DATETIME**, **TIMEDELTA**, **OBJECT**, **STRING**, **VOID**

**INTP**, **UINTP**

**GENBOOL**, **SIGNED**, **UNSIGNED**, **FLOATING**, **COMPLEX**

The latter group of `{NAME}s` corresponds to letters used in the array interface typestring specification.

## 8.3.2 Defines

### Max and min values for integers

**`NPY_MAX_INT{bits}`**

**`NPY_MAX_UINT{bits}`**

**`NPY_MIN_INT{bits}`**

These are defined for `{bits}` = 8, 16, 32, 64, 128, and 256 and provide the maximum (minimum) value of the corresponding (unsigned) integer type. Note: the actual integer type may not be available on all platforms (i.e. 128-bit and 256-bit integers are rare).

**`NPY_MIN_{type}`**

This is defined for `{type}` = **BYTE**, **SHORT**, **INT**, **LONG**, **LONGLONG**, **INTP**

**`NPY_MAX_{type}`**

This is defined for all defined for `{type}` = **BYTE**, **UBYTE**, **SHORT**, **USHORT**, **INT**, **UINT**, **LONG**, **ULONG**, **LONGLONG**, **ULONGLONG**, **INTP**, **UINTP**

### Number of bits in data types

All `NPY_SIZEOF_{CTYPE}` constants have corresponding `NPY_BITSOF_{CTYPE}` constants defined. The `NPY_BITSOF_{CTYPE}` constants provide the number of bits in the data type. Specifically, the available `{CTYPE}s` are

> **BOOL**, **CHAR**, **SHORT**, **INT**, **LONG**, **LONGLONG**, **FLOAT**, **DOUBLE**, **LONGDOUBLE**

### Bit-width references to enumerated typenums

All of the numeric data types (integer, floating point, and complex) have constants that are defined to be a specific enumerated type number. Exactly which enumerated type a bit-width type refers to is platform dependent. In particular, the constants available are `PyArray_{NAME}{BITS}` where `{NAME}` is **INT**, **UINT**, **FLOAT**, **COMPLEX** and `{BITS}` can be 8, 16, 32, 64, 80, 96, 128, 160, 192, 256, and 512. Obviously not all bit-widths are available on all platforms for all the kinds of numeric types. Commonly 8-, 16-, 32-, 64-bit integers; 32-, 64-bit floats; and 64-, 128-bit complex types are available.

### Integer that can hold a pointer

The constants **NPY_INTP** and **NPY_UINTP** refer to an enumerated integer type that is large enough to hold a pointer on the platform. Index arrays should always be converted to **NPY_INTP** , because the dimension of the array is of type npy_intp.

## 8.3.3 C-type names

There are standard variable types for each of the numeric data types and the bool data type. Some of these are already available in the C-specification. You can create variables in extension code with these types.

### Boolean

**`npy_bool`**
unsigned char; The constants *`NPY_FALSE`* and *`NPY_TRUE`* are also defined.

### (Un)Signed Integer

Unsigned versions of the integers can be defined by pre-pending a 'u' to the front of the integer name.

**`npy_(u)byte`**
(unsigned) char

**`npy_short`**
short

**`npy_ushort`**
unsigned short

**`npy_uint`**
unsigned int

**`npy_int`**
int

**`npy_int16`**
16-bit integer

**`npy_uint16`**
16-bit unsigned integer

**`npy_int32`**
32-bit integer

**`npy_uint32`**
32-bit unsigned integer

**`npy_int64`**
64-bit integer

**`npy_uint64`**
64-bit unsigned integer

**`npy_(u)long`**
(unsigned) long int

**`npy_(u)longlong`**
(unsigned long long int)

**`npy_intp`**
Py_intptr_t (an integer that is the size of a pointer on the platform).

**`npy_uintp`**
unsigned Py_intptr_t (an integer that is the size of a pointer on the platform).

### (Complex) Floating point

**`npy_half`**
16-bit float

**`npy_(c)float`**
32-bit float

**`npy_(c)double`**
64-bit double

**`npy_(c)longdouble`**
long double

complex types are structures with **.real** and **.imag** members (in that order).

### Bit-width names

There are also typedefs for signed integers, unsigned integers, floating point, and complex floating point types of specific bit- widths. The available type names are

> `npy_int{bits}`, `npy_uint{bits}`, `npy_float{bits}`, and `npy_complex{bits}`

where `{bits}` is the number of bits in the type and can be **8**, **16**, **32**, **64**, 128, and 256 for integer types; 16, **32** , **64**, 80, 96, 128, and 256 for floating-point types; and 32, **64**, **128**, 160, 192, and 512 for complex-valued types. Which bit-widths are available is platform dependent. The bolded bit-widths are usually available on all platforms.

## 8.3.4 Printf Formatting

For help in printing, the following strings are defined as the correct format specifier in printf and related commands.

> `NPY_LONGLONG_FMT`, `NPY_ULONGLONG_FMT`, `NPY_INTP_FMT`, `NPY_UINTP_FMT`, `NPY_LONGDOUBLE_FMT`

# 8.4 Array API

The test of a first-rate intelligence is the ability to hold two
opposed ideas in the mind at the same time, and still retain the
ability to function.
— *F. Scott Fitzgerald*

For a successful technology, reality must take precedence over public
relations, for Nature cannot be fooled.
— *Richard P. Feynman*

### 8.4.1 Array structure and data access

These macros access the `PyArrayObject` structure members and are defined in `ndarraytypes.h`. The input argument, *arr*, can be any `PyObject *` that is directly interpretable as a `PyArrayObject *` (any instance of the `PyArray_Type` and itssub-types).

int **PyArray_NDIM**(*PyArrayObject *arr*)

The number of dimensions in the array.

int **PyArray_FLAGS**(*PyArrayObject* arr*)

Returns an integer representing the *array-flags*.

int **PyArray_TYPE**(*PyArrayObject* arr*)

Return the (builtin) typenumber for the elements of this array.

int **PyArray_SETITEM**(*PyArrayObject* arr*, void* *itemptr*, PyObject* *obj*)

Convert obj and place it in the ndarray, *arr*, at the place pointed to by itemptr. Return -1 if an error occurs or 0 on success.

void **PyArray_ENABLEFLAGS**(*PyArrayObject* arr*, int *flags*)

New in version 1.7.

Enables the specified array flags. This function does no validation, and assumes that you know what you're doing.

void **PyArray_CLEARFLAGS**(*PyArrayObject* arr*, int *flags*)

New in version 1.7.

Clears the specified array flags. This function does no validation, and assumes that you know what you're doing.

void ***PyArray_DATA**(*PyArrayObject *arr*)

char ***PyArray_BYTES**(*PyArrayObject *arr*)

These two macros are similar and obtain the pointer to the data-buffer for the array. The first macro can (and should be) assigned to a particular pointer where the second is for generic processing. If you have not guaranteed a contiguous and/or aligned array then be sure you understand how to access the data in the array to avoid memory and/or alignment problems.

*npy_intp* ***PyArray_DIMS**(*PyArrayObject *arr*)

Returns a pointer to the dimensions/shape of the array. The number of elements matches the number of dimensions of the array. Can return `NULL` for 0-dimensional arrays.

*npy_intp* ***PyArray_SHAPE**(*PyArrayObject *arr*)

New in version 1.7.

A synonym for `PyArray_DIMS`, named to be consistent with the `shape` usage within Python.

*npy_intp* ***PyArray_STRIDES**(*PyArrayObject* arr*)

Returns a pointer to the strides of the array. The number of elements matches the number of dimensions of the array.

*npy_intp* **PyArray_DIM**(*PyArrayObject* arr*, int *n*)

Return the shape in the $n^{\text{th}}$ dimension.

*npy_intp* **PyArray_STRIDE**(*PyArrayObject* arr*, int *n*)

Return the stride in the $n^{\text{th}}$ dimension.

*npy_intp* **PyArray_ITEMSIZE**(*PyArrayObject* arr*)

Return the itemsize for the elements of this array.

Note that, in the old API that was deprecated in version 1.7, this function had the return type `int`.

*npy_intp* **PyArray_SIZE**(*PyArrayObject* arr*)

Returns the total size (in number of elements) of the array.

*npy_intp* **`PyArray_Size`** (*PyArrayObject* obj*)
Returns 0 if *obj* is not a sub-class of ndarray. Otherwise, returns the total number of elements in the array. Safer version of `PyArray_SIZE` (*obj*).

*npy_intp* **`PyArray_NBYTES`** (*PyArrayObject* arr*)
Returns the total number of bytes consumed by the array.

PyObject ***`PyArray_BASE`** (*PyArrayObject* arr*)
This returns the base object of the array. In most cases, this means the object which owns the memory the array is pointing at.

If you are constructing an array using the C API, and specifying your own memory, you should use the function `PyArray_SetBaseObject` to set the base to an object which owns the memory.

If the (deprecated) `NPY_ARRAY_UPDATEIFCOPY` or the `NPY_ARRAY_WRITEBACKIFCOPY` flags are set, it has a different meaning, namely base is the array into which the current array will be copied upon copy resolution. This overloading of the base property for two functions is likely to change in a future version of NumPy.

*PyArray_Descr* ***`PyArray_DESCR`** (*PyArrayObject* arr*)
Returns a borrowed reference to the dtype property of the array.

*PyArray_Descr* ***`PyArray_DTYPE`** (*PyArrayObject* arr*)
New in version 1.7.

A synonym for PyArray_DESCR, named to be consistent with the 'dtype' usage within Python.

PyObject ***`PyArray_GETITEM`** (*PyArrayObject* arr*, void* *itemptr*)
Get a Python object of a builtin type from the ndarray, *arr*, at the location pointed to by itemptr. Return `NULL` on failure.

`numpy.ndarray.item` is identical to PyArray_GETITEM.

### Data access

These functions and macros provide easy access to elements of the ndarray from C. These work for all arrays. You may need to take care when accessing the data in the array, however, if it is not in machine byte-order, misaligned, or not writeable. In other words, be sure to respect the state of the flags unless you know what you are doing, or have previously guaranteed an array that is writeable, aligned, and in machine byte-order using `PyArray_FromAny`. If you wish to handle all types of arrays, the copyswap function for each type is useful for handling misbehaved arrays. Some platforms (e.g. Solaris) do not like misaligned data and will crash if you de-reference a misaligned pointer. Other platforms (e.g. x86 Linux) will just work more slowly with misaligned data.

void* **`PyArray_GetPtr`** (*PyArrayObject* aobj*, *npy_intp* ind*)
Return a pointer to the data of the ndarray, *aobj*, at the N-dimensional index given by the c-array, *ind*, (which must be at least *aobj* ->nd in size). You may want to typecast the returned pointer to the data type of the ndarray.

void* **`PyArray_GETPTR1`** (*PyArrayObject* obj*, *npy_intp i*)

void* **`PyArray_GETPTR2`** (*PyArrayObject* obj*, *npy_intp i*, *npy_intp j*)

void* **`PyArray_GETPTR3`** (*PyArrayObject* obj*, *npy_intp i*, *npy_intp j*, *npy_intp k*)

void* **`PyArray_GETPTR4`** (*PyArrayObject* obj*, *npy_intp i*, *npy_intp j*, *npy_intp k*, *npy_intp l*)
Quick, inline access to the element at the given coordinates in the ndarray, *obj*, which must have respectively 1, 2, 3, or 4 dimensions (this is not checked). The corresponding *i*, *j*, *k*, and *l* coordinates can be any integer but will be interpreted as `npy_intp`. You may want to typecast the returned pointer to the data type of the ndarray.

## 8.4.2 Creating arrays

### From scratch

PyObject* **PyArray_NewFromDescr** (PyTypeObject* *subtype*, PyArray_Descr* *descr*, int *nd*, npy_intp const* *dims*, npy_intp const* *strides*, void* *data*, int *flags*, PyObject* *obj*)

This function steals a reference to *descr*. The easiest way to get one is using `PyArray_DescrFromType`.

This is the main array creation function. Most new arrays are created with this flexible function.

The returned object is an object of Python-type *subtype*, which must be a subtype of `PyArray_Type`. The array has *nd* dimensions, described by *dims*. The data-type descriptor of the new array is *descr*.

If *subtype* is of an array subclass instead of the base `&PyArray_Type`, then *obj* is the object to pass to the `__array_finalize__` method of the subclass.

If *data* is `NULL`, then new unitinialized memory will be allocated and *flags* can be non-zero to indicate a Fortran-style contiguous array. Use `PyArray_FILLWBYTE` to initialize the memory.

If *data* is not `NULL`, then it is assumed to point to the memory to be used for the array and the *flags* argument is used as the new flags for the array (except the state of `NPY_ARRAY_OWNDATA`, `NPY_ARRAY_WRITEBACKIFCOPY` and `NPY_ARRAY_UPDATEIFCOPY` flags of the new array will be reset).

In addition, if *data* is non-NULL, then *strides* can also be provided. If *strides* is `NULL`, then the array strides are computed as C-style contiguous (default) or Fortran-style contiguous (*flags* is nonzero for *data* = `NULL` or *flags* & `NPY_ARRAY_F_CONTIGUOUS` is nonzero non-NULL *data*). Any provided *dims* and *strides* are copied into newly allocated dimension and strides arrays for the new array object.

`PyArray_CheckStrides` can help verify non- `NULL` stride information.

If `data` is provided, it must stay alive for the life of the array. One way to manage this is through `PyArray_SetBaseObject`

PyObject* **PyArray_NewLikeArray** (PyArrayObject* *prototype*, NPY_ORDER *order*, PyArray_Descr* *descr*, int *subok*)

New in version 1.6.

This function steals a reference to *descr* if it is not NULL.

This array creation routine allows for the convenient creation of a new array matching an existing array's shapes and memory layout, possibly changing the layout and/or data type.

When *order* is `NPY_ANYORDER`, the result order is `NPY_FORTRANORDER` if *prototype* is a fortran array, `NPY_CORDER` otherwise. When *order* is `NPY_KEEPORDER`, the result order matches that of *prototype*, even when the axes of *prototype* aren't in C or Fortran order.

If *descr* is NULL, the data type of *prototype* is used.

If *subok* is 1, the newly created array will use the sub-type of *prototype* to create the new array, otherwise it will create a base-class array.

PyObject* **PyArray_New** (PyTypeObject* *subtype*, int *nd*, npy_intp const* *dims*, int *type_num*, npy_intp const* *strides*, void* *data*, int *itemsize*, int *flags*, PyObject* *obj*)

This is similar to `PyArray_NewFromDescr` (…) except you specify the data-type descriptor with *type_num* and *itemsize*, where *type_num* corresponds to a builtin (or user-defined) type. If the type always has the same number of bytes, then itemsize is ignored. Otherwise, itemsize specifies the particular size of this array.

**Warning:** If data is passed to `PyArray_NewFromDescr` or `PyArray_New`, this memory must not be deallocated until the new array is deleted. If this data came from another Python object, this can be accomplished using

`Py_INCREF` on that object and setting the base member of the new array to point to that object. If strides are passed in they must be consistent with the dimensions, the itemsize, and the data of the array.

PyObject* **PyArray_SimpleNew** (int *nd*, *npy_intp* const* *dims*, int *typenum*)

Create a new uninitialized array of type, *typenum*, whose size in each of *nd* dimensions is given by the integer array, *dims*.The memory for the array is uninitialized (unless typenum is `NPY_OBJECT` in which case each element in the array is set to NULL). The *typenum* argument allows specification of any of the builtin data-types such as `NPY_FLOAT` or `NPY_LONG`. The memory for the array can be set to zero if desired using `PyArray_FILLWBYTE` (return_object, 0).This function cannot be used to create a flexible-type array (no itemsize given).

PyObject* **PyArray_SimpleNewFromData** (int *nd*, *npy_intp* const* *dims*, int *typenum*, void* *data*)

Create an array wrapper around *data* pointed to by the given pointer. The array flags will have a default that the data area is well-behaved and C-style contiguous. The shape of the array is given by the *dims* c-array of length *nd*. The data-type of the array is indicated by *typenum*. If data comes from another reference-counted Python object, the reference count on this object should be increased after the pointer is passed in, and the base member of the returned ndarray should point to the Python object that owns the data. This will ensure that the provided memory is not freed while the returned array is in existence. To free memory as soon as the ndarray is deallocated, set the OWNDATA flag on the returned ndarray.

PyObject* **PyArray_SimpleNewFromDescr** (int *nd*, *npy_int* const* *dims*, *PyArray_Descr** *descr*)

This function steals a reference to *descr*.

Create a new array with the provided data-type descriptor, *descr*, of the shape determined by *nd* and *dims*.

**PyArray_FILLWBYTE** (PyObject* *obj*, int *val*)

Fill the array pointed to by *obj* —which must be a (subclass of) ndarray—with the contents of *val* (evaluated as a byte). This macro calls memset, so obj must be contiguous.

PyObject* **PyArray_Zeros** (int *nd*, *npy_intp* const* *dims*, *PyArray_Descr** *dtype*, int *fortran*)

Construct a new *nd* -dimensional array with shape given by *dims* and data type given by *dtype*. If *fortran* is non-zero, then a Fortran-order array is created, otherwise a C-order array is created. Fill the memory with zeros (or the 0 object if *dtype* corresponds to `NPY_OBJECT` ).

PyObject* **PyArray_ZEROS** (int *nd*, *npy_intp* const* *dims*, int *type_num*, int *fortran*)

Macro form of `PyArray_Zeros` which takes a type-number instead of a data-type object.

PyObject* **PyArray_Empty** (int *nd*, *npy_intp* const* *dims*, *PyArray_Descr** *dtype*, int *fortran*)

Construct a new *nd* -dimensional array with shape given by *dims* and data type given by *dtype*. If *fortran* is non-zero, then a Fortran-order array is created, otherwise a C-order array is created. The array is uninitialized unless the data type corresponds to `NPY_OBJECT` in which case the array is filled with `Py_None`.

PyObject* **PyArray_EMPTY** (int *nd*, *npy_intp* const* *dims*, int *typenum*, int *fortran*)

Macro form of `PyArray_Empty` which takes a type-number, *typenum*, instead of a data-type object.

PyObject* **PyArray_Arange** (double *start*, double *stop*, double *step*, int *typenum*)

Construct a new 1-dimensional array of data-type, *typenum*, that ranges from *start* to *stop* (exclusive) in increments of *step* . Equivalent to **arange** (*start*, *stop*, *step*, dtype).

PyObject* **PyArray_ArangeObj** (PyObject* *start*, PyObject* *stop*, PyObject* *step*, *PyArray_Descr** *descr*)

Construct a new 1-dimensional array of data-type determined by `descr`, that ranges from `start` to `stop` (exclusive) in increments of `step`. Equivalent to arange( `start`, `stop`, `step`, `typenum` ).

int **PyArray_SetBaseObject** (*PyArrayObject** *arr*, PyObject* *obj*)

New in version 1.7.

This function **steals a reference** to `obj` and sets it as the base property of `arr`.

If you construct an array by passing in your own memory buffer as a parameter, you need to set the array's *base* property to ensure the lifetime of the memory buffer is appropriate.

The return value is 0 on success, -1 on failure.

If the object provided is an array, this function traverses the chain of *base* pointers so that each array points to the owner of the memory directly. Once the base is set, it may not be changed to another value.

### From other objects

PyObject* **`PyArray_FromAny`**(PyObject* *op*, *PyArray_Descr** *dtype*, int *min_depth*, int *max_depth*, int *requirements*, PyObject* *context*)

This is the main function used to obtain an array from any nested sequence, or object that exposes the array interface, *op*. The parameters allow specification of the required *dtype*, the minimum (*min_depth*) and maximum (*max_depth*) number of dimensions acceptable, and other *requirements* for the array. This function **steals a reference** to the dtype argument, which needs to be a *`PyArray_Descr`* structure indicating the desired data-type (including required byteorder). The *dtype* argument may be `NULL`, indicating that any data-type (and byteorder) is acceptable. Unless *`NPY_ARRAY_FORCECAST`* is present in `flags`, this call will generate an error if the data type cannot be safely obtained from the object. If you want to use `NULL` for the *dtype* and ensure the array is notswapped then use *`PyArray_CheckFromAny`*. A value of 0 for either of the depth parameters causes the parameter to be ignored. Any of the following array flags can be added (*e.g.* using |) to get the *requirements* argument. If your code can handle general (*e.g.* strided, byte-swapped, or unaligned arrays) then *requirements* may be 0. Also, if *op* is not already an array (or does not expose the array interface), then a new array will be created (and filled from *op* using the sequence protocol). The new array will have *`NPY_ARRAY_DEFAULT`* as its flags member. The *context* argument is unused.

**`NPY_ARRAY_C_CONTIGUOUS`**
Make sure the returned array is C-style contiguous

**`NPY_ARRAY_F_CONTIGUOUS`**
Make sure the returned array is Fortran-style contiguous.

**`NPY_ARRAY_ALIGNED`**
Make sure the returned array is aligned on proper boundaries for its data type. An aligned array has the data pointer and every strides factor as a multiple of the alignment factor for the data-type- descriptor.

**`NPY_ARRAY_WRITEABLE`**
Make sure the returned array can be written to.

**`NPY_ARRAY_ENSURECOPY`**
Make sure a copy is made of *op*. If this flag is not present, data is not copied if it can be avoided.

**`NPY_ARRAY_ENSUREARRAY`**
Make sure the result is a base-class ndarray. By default, if *op* is an instance of a subclass of ndarray, an instance of that same subclass is returned. If this flag is set, an ndarray object will be returned instead.

**`NPY_ARRAY_FORCECAST`**
Force a cast to the output type even if it cannot be done safely. Without this flag, a data cast will occur only if it can be done safely, otherwise an error is raised.

**`NPY_ARRAY_WRITEBACKIFCOPY`**
If *op* is already an array, but does not satisfy the requirements, then a copy is made (which will satisfy the requirements). If this flag is present and a copy (of an object that is already an array) must be made, then the corresponding *`NPY_ARRAY_WRITEBACKIFCOPY`* flag is set in the returned copy and *op* is made to be read-only. You must be sure to call *`PyArray_ResolveWritebackIfCopy`* to copy the contents back into *op* and the *op* array will be made writeable again. If *op* is not writeable to begin with, or if it is not already an array, then an error is raised.

**NPY_ARRAY_UPDATEIFCOPY**
Deprecated. Use *NPY_ARRAY_WRITEBACKIFCOPY*, which is similar. This flag "automatically" copies the data back when the returned array is deallocated, which is not supported in all python implementations.

**NPY_ARRAY_BEHAVED**
*NPY_ARRAY_ALIGNED* | *NPY_ARRAY_WRITEABLE*

**NPY_ARRAY_CARRAY**
*NPY_ARRAY_C_CONTIGUOUS* | *NPY_ARRAY_BEHAVED*

**NPY_ARRAY_CARRAY_RO**
*NPY_ARRAY_C_CONTIGUOUS* | *NPY_ARRAY_ALIGNED*

**NPY_ARRAY_FARRAY**
*NPY_ARRAY_F_CONTIGUOUS* | *NPY_ARRAY_BEHAVED*

**NPY_ARRAY_FARRAY_RO**
*NPY_ARRAY_F_CONTIGUOUS* | *NPY_ARRAY_ALIGNED*

**NPY_ARRAY_DEFAULT**
*NPY_ARRAY_CARRAY*

**NPY_ARRAY_IN_ARRAY**
*NPY_ARRAY_C_CONTIGUOUS* | *NPY_ARRAY_ALIGNED*

**NPY_ARRAY_IN_FARRAY**
*NPY_ARRAY_F_CONTIGUOUS* | *NPY_ARRAY_ALIGNED*

**NPY_OUT_ARRAY**
*NPY_ARRAY_C_CONTIGUOUS* | *NPY_ARRAY_WRITEABLE* | *NPY_ARRAY_ALIGNED*

**NPY_ARRAY_OUT_ARRAY**
*NPY_ARRAY_C_CONTIGUOUS* | *NPY_ARRAY_ALIGNED* | *NPY_ARRAY_WRITEABLE*

**NPY_ARRAY_OUT_FARRAY**
*NPY_ARRAY_F_CONTIGUOUS* | *NPY_ARRAY_WRITEABLE* | *NPY_ARRAY_ALIGNED*

**NPY_ARRAY_INOUT_ARRAY**
*NPY_ARRAY_C_CONTIGUOUS* | *NPY_ARRAY_WRITEABLE* | *NPY_ARRAY_ALIGNED* | *NPY_ARRAY_WRITEBACKIFCOPY* | *NPY_ARRAY_UPDATEIFCOPY*

**NPY_ARRAY_INOUT_FARRAY**
*NPY_ARRAY_F_CONTIGUOUS* | *NPY_ARRAY_WRITEABLE* | *NPY_ARRAY_ALIGNED* | *NPY_ARRAY_WRITEBACKIFCOPY* | *NPY_ARRAY_UPDATEIFCOPY*

int **PyArray_GetArrayParamsFromObject** (PyObject* *op*, *PyArray_Descr** *requested_dtype*, *npy_bool* *writeable*, *PyArray_Descr*** *out_dtype*, int* *out_ndim*, *npy_intp** *out_dims*, *PyArrayObject*** *out_arr*, PyObject* *context*)

Deprecated since version NumPy: 1.19

Unless NumPy is made aware of an issue with this, this function is scheduled for rapid removal without replacement.

Changed in version NumPy: 1.19

*context* is never used. Its use results in an error.

New in version 1.6.

PyObject* **PyArray_CheckFromAny** (PyObject* *op*, *PyArray_Descr** *dtype*, int *min_depth*, int *max_depth*, int *requirements*, PyObject* *context*)
Nearly identical to *PyArray_FromAny* (…) except *requirements* can contain *NPY_ARRAY_NOTSWAPPED* (over-riding the specification in *dtype*) and *NPY_ARRAY_ELEMENTSTRIDES* which indicates that the array should be aligned in the sense that the strides are multiples of the element size.

In versions 1.6 and earlier of NumPy, the following flags did not have the _ARRAY_ macro namespace in them. That form of the constant names is deprecated in 1.7.

**`NPY_ARRAY_NOTSWAPPED`**
Make sure the returned array has a data-type descriptor that is in machine byte-order, over-riding any specification in the *dtype* argument. Normally, the byte-order requirement is determined by the *dtype* argument. If this flag is set and the dtype argument does not indicate a machine byte-order descriptor (or is NULL and the object is already an array with a data-type descriptor that is not in machine byte- order), then a new data-type descriptor is created and used with its byte-order field set to native.

**`NPY_ARRAY_BEHAVED_NS`**
`NPY_ARRAY_ALIGNED` | `NPY_ARRAY_WRITEABLE` | `NPY_ARRAY_NOTSWAPPED`

**`NPY_ARRAY_ELEMENTSTRIDES`**
Make sure the returned array has strides that are multiples of the element size.

PyObject* **`PyArray_FromArray`**(*PyArrayObject* op*, *PyArray_Descr* newtype*, int *requirements*)
Special case of `PyArray_FromAny` for when *op* is already an array but it needs to be of a specific *newtype* (including byte-order) or has certain *requirements*.

PyObject* **`PyArray_FromStructInterface`**(PyObject* *op*)
Returns an ndarray object from a Python object that exposes the `__array_struct__` attribute and follows the array interface protocol. If the object does not contain this attribute then a borrowed reference to `Py_NotImplemented` is returned.

PyObject* **`PyArray_FromInterface`**(PyObject* *op*)
Returns an ndarray object from a Python object that exposes the `__array_interface__` attribute following the array interface protocol. If the object does not contain this attribute then a borrowed reference to `Py_NotImplemented` is returned.

PyObject* **`PyArray_FromArrayAttr`**(PyObject* *op*, *PyArray_Descr* dtype*, PyObject* *context*)
Return an ndarray object from a Python object that exposes the `__array__` method. The `__array__` method can take 0, or 1 argument `([dtype])`. `context` is unused.

PyObject* **`PyArray_ContiguousFromAny`**(PyObject* *op*, int *typenum*, int *min_depth*, int *max_depth*)
This function returns a (C-style) contiguous and behaved function array from any nested sequence or array interface exporting object, *op*, of (non-flexible) type given by the enumerated *typenum*, of minimum depth *min_depth*, and of maximum depth *max_depth*. Equivalent to a call to `PyArray_FromAny` with requirements set to `NPY_ARRAY_DEFAULT` and the type_num member of the type argument set to *typenum*.

PyObject ***`PyArray_FromObject`**(PyObject **op*, int *typenum*, int *min_depth*, int *max_depth*)
Return an aligned and in native-byteorder array from any nested sequence or array-interface exporting object, op, of a type given by the enumerated typenum. The minimum number of dimensions the array can have is given by min_depth while the maximum is max_depth. This is equivalent to a call to `PyArray_FromAny` with requirements set to BEHAVED.

PyObject* **`PyArray_EnsureArray`**(PyObject* *op*)
This function **steals a reference** to `op` and makes sure that `op` is a base-class ndarray. It special cases array scalars, but otherwise calls `PyArray_FromAny` ( `op`, NULL, 0, 0, `NPY_ARRAY_ENSUREARRAY`, NULL).

PyObject* **`PyArray_FromString`**(char* *string*, *npy_intp slen*, *PyArray_Descr* dtype*, *npy_intp num*, char* *sep*)
Construct a one-dimensional ndarray of a single type from a binary or (ASCII) text `string` of length `slen`. The data-type of the array to-be-created is given by `dtype`. If num is -1, then **copy** the entire string and return an appropriately sized array, otherwise, `num` is the number of items to **copy** from the string. If `sep` is NULL (or “”), then interpret the string as bytes of binary data, otherwise convert the sub-strings separated by `sep` to items of data-type `dtype`. Some data-types may not be readable in text mode and an error will be raised if that occurs. All errors return NULL.

PyObject* **PyArray_FromFile** (FILE* *fp*, *PyArray_Descr** *dtype*, *npy_intp* *num*, char* *sep*)

Construct a one-dimensional ndarray of a single type from a binary or text file. The open file pointer is `fp`, the data-type of the array to be created is given by `dtype`. This must match the data in the file. If `num` is -1, then read until the end of the file and return an appropriately sized array, otherwise, `num` is the number of items to read. If `sep` is NULL (or ""), then read from the file in binary mode, otherwise read from the file in text mode with `sep` providing the item separator. Some array types cannot be read in text mode in which case an error is raised.

PyObject* **PyArray_FromBuffer** (PyObject* *buf*, *PyArray_Descr** *dtype*, *npy_intp* *count*, *npy_intp* *offset*)

Construct a one-dimensional ndarray of a single type from an object, `buf`, that exports the (single-segment) buffer protocol (or has an attribute __buffer__ that returns an object that exports the buffer protocol). A writeable buffer will be tried first followed by a read- only buffer. The *`NPY_ARRAY_WRITEABLE`* flag of the returned array will reflect which one was successful. The data is assumed to start at `offset` bytes from the start of the memory location for the object. The type of the data in the buffer will be interpreted depending on the data- type descriptor, `dtype.` If `count` is negative then it will be determined from the size of the buffer and the requested itemsize, otherwise, `count` represents how many elements should be converted from the buffer.

int **PyArray_CopyInto** (*PyArrayObject** *dest*, *PyArrayObject** *src*)

Copy from the source array, `src`, into the destination array, `dest`, performing a data-type conversion if necessary. If an error occurs return -1 (otherwise 0). The shape of `src` must be broadcastable to the shape of `dest`. The data areas of dest and src must not overlap.

int **PyArray_MoveInto** (*PyArrayObject** *dest*, *PyArrayObject** *src*)

Move data from the source array, `src`, into the destination array, `dest`, performing a data-type conversion if necessary. If an error occurs return -1 (otherwise 0). The shape of `src` must be broadcastable to the shape of `dest`. The data areas of dest and src may overlap.

*PyArrayObject** **PyArray_GETCONTIGUOUS** (PyObject* *op*)

If `op` is already (C-style) contiguous and well-behaved then just return a reference, otherwise return a (contiguous and well-behaved) copy of the array. The parameter op must be a (sub-class of an) ndarray and no checking for that is done.

PyObject* **PyArray_FROM_O** (PyObject* *obj*)

Convert `obj` to an ndarray. The argument can be any nested sequence or object that exports the array interface. This is a macro form of *`PyArray_FromAny`* using `NULL`, 0, 0, 0 for the other arguments. Your code must be able to handle any data-type descriptor and any combination of data-flags to use this macro.

PyObject* **PyArray_FROM_OF** (PyObject* *obj*, int *requirements*)

Similar to *`PyArray_FROM_O`* except it can take an argument of *requirements* indicating properties the resulting array must have. Available requirements that can be enforced are *`NPY_ARRAY_C_CONTIGUOUS`*, *`NPY_ARRAY_F_CONTIGUOUS`*, *`NPY_ARRAY_ALIGNED`*, *`NPY_ARRAY_WRITEABLE`*, *`NPY_ARRAY_NOTSWAPPED`*, *`NPY_ARRAY_ENSURECOPY`*, *`NPY_ARRAY_WRITEBACKIFCOPY`*, *`NPY_ARRAY_UPDATEIFCOPY`*, *`NPY_ARRAY_FORCECAST`*, and *`NPY_ARRAY_ENSUREARRAY`*. Standard combinations of flags can also be used:

PyObject* **PyArray_FROM_OT** (PyObject* *obj*, int *typenum*)

Similar to *`PyArray_FROM_O`* except it can take an argument of *typenum* specifying the type-number the returned array.

PyObject* **PyArray_FROM_OTF** (PyObject* *obj*, int *typenum*, int *requirements*)

Combination of *`PyArray_FROM_OF`* and *`PyArray_FROM_OT`* allowing both a *typenum* and a *flags* argument to be provided.

PyObject* **PyArray_FROMANY** (PyObject* *obj*, int *typenum*, int *min*, int *max*, int *requirements*)

Similar to *`PyArray_FromAny`* except the data-type is specified using a typenumber. *`PyArray_DescrFromType`* (*typenum*) is passed directly to *`PyArray_FromAny`*. This macro also adds *`NPY_ARRAY_DEFAULT`* to requirements if *`NPY_ARRAY_ENSURECOPY`* is passed in as requirements.

PyObject ***PyArray_CheckAxis** (PyObject* *obj*, int* *axis*, int *requirements*)

Encapsulate the functionality of functions and methods that take the axis= keyword and work properly with None as the axis argument. The input array is `obj`, while `*axis` is a converted integer (so that >=MAXDIMS is the None value), and `requirements` gives the needed properties of `obj`. The output is a converted version of the input so that requirements are met and if needed a flattening has occurred. On output negative values of `*axis` are converted and the new value is checked to ensure consistency with the shape of `obj`.

### 8.4.3 Dealing with types

#### General check of Python Type

**`PyArray_Check`** (PyObject **op*)
Evaluates true if *op* is a Python object whose type is a sub-type of *`PyArray_Type`*.

**`PyArray_CheckExact`** (PyObject **op*)
Evaluates true if *op* is a Python object with type *`PyArray_Type`*.

**`PyArray_HasArrayInterface`** (PyObject **op*, PyObject **out*)
If `op` implements any part of the array interface, then `out` will contain a new reference to the newly created ndarray using the interface or `out` will contain `NULL` if an error during conversion occurs. Otherwise, out will contain a borrowed reference to `Py_NotImplemented` and no error condition is set.

**`PyArray_HasArrayInterfaceType`** (op, dtype, context, out)
If `op` implements any part of the array interface, then `out` will contain a new reference to the newly created ndarray using the interface or `out` will contain `NULL` if an error during conversion occurs. Otherwise, out will contain a borrowed reference to Py_NotImplemented and no error condition is set. This version allows setting of the dtype in the part of the array interface that looks for the *`__array__`* attribute. *context* is unused.

**`PyArray_IsZeroDim`** (op)
Evaluates true if *op* is an instance of (a subclass of) *`PyArray_Type`* and has 0 dimensions.

**`PyArray_IsScalar`** (op, cls)
Evaluates true if *op* is an instance of `Py{cls}ArrType_Type`.

**`PyArray_CheckScalar`** (op)
Evaluates true if *op* is either an array scalar (an instance of a sub-type of `PyGenericArr_Type` ), or an instance of (a sub-class of) *`PyArray_Type`* whose dimensionality is 0.

**`PyArray_IsPythonNumber`** (op)
Evaluates true if *op* is an instance of a builtin numeric type (int, float, complex, long, bool)

**`PyArray_IsPythonScalar`** (op)
Evaluates true if *op* is a builtin Python scalar object (int, float, complex, bytes, str, long, bool).

**`PyArray_IsAnyScalar`** (op)
Evaluates true if *op* is either a Python scalar object (see *`PyArray_IsPythonScalar`*) or an array scalar (an instance of a sub- type of `PyGenericArr_Type` ).

**`PyArray_CheckAnyScalar`** (op)
Evaluates true if *op* is a Python scalar object (see *`PyArray_IsPythonScalar`*), an array scalar (an instance of a sub-type of `PyGenericArr_Type`) or an instance of a sub-type of *`PyArray_Type`* whose dimensionality is 0.

### Data-type checking

For the typenum macros, the argument is an integer representing an enumerated array data type. For the array type checking macros the argument must be a `PyObject *` that can be directly interpreted as a `PyArrayObject *`.

**`PyTypeNum_ISUNSIGNED`** (int *num*)

**`PyDataType_ISUNSIGNED`** (*PyArray_Descr *descr*)

**`PyArray_ISUNSIGNED`** (*PyArrayObject *obj*)
Type represents an unsigned integer.

**`PyTypeNum_ISSIGNED`** (int *num*)

**`PyDataType_ISSIGNED`** (*PyArray_Descr *descr*)

**`PyArray_ISSIGNED`** (*PyArrayObject *obj*)
Type represents a signed integer.

**`PyTypeNum_ISINTEGER`** (int *num*)

**`PyDataType_ISINTEGER`** (*PyArray_Descr* descr*)

**`PyArray_ISINTEGER`** (*PyArrayObject *obj*)
Type represents any integer.

**`PyTypeNum_ISFLOAT`** (int *num*)

**`PyDataType_ISFLOAT`** (*PyArray_Descr* descr*)

**`PyArray_ISFLOAT`** (*PyArrayObject *obj*)
Type represents any floating point number.

**`PyTypeNum_ISCOMPLEX`** (int *num*)

**`PyDataType_ISCOMPLEX`** (*PyArray_Descr* descr*)

**`PyArray_ISCOMPLEX`** (*PyArrayObject *obj*)
Type represents any complex floating point number.

**`PyTypeNum_ISNUMBER`** (int *num*)

**`PyDataType_ISNUMBER`** (*PyArray_Descr* descr*)

**`PyArray_ISNUMBER`** (*PyArrayObject *obj*)
Type represents any integer, floating point, or complex floating point number.

**`PyTypeNum_ISSTRING`** (int *num*)

**`PyDataType_ISSTRING`** (*PyArray_Descr* descr*)

**`PyArray_ISSTRING`** (*PyArrayObject *obj*)
Type represents a string data type.

**`PyTypeNum_ISPYTHON`** (int *num*)

**`PyDataType_ISPYTHON`** (*PyArray_Descr* descr*)

**`PyArray_ISPYTHON`** (*PyArrayObject *obj*)
Type represents an enumerated type corresponding to one of the standard Python scalar (bool, int, float, or complex).

**`PyTypeNum_ISFLEXIBLE`** (int *num*)

**`PyDataType_ISFLEXIBLE`** (*PyArray_Descr* descr*)

**PyArray_ISFLEXIBLE** (*PyArrayObject *obj*)
Type represents one of the flexible array types ( `NPY_STRING`, `NPY_UNICODE`, or `NPY_VOID` ).

**PyDataType_ISUNSIZED(PyArray_Descr* descr):**
Type has no size information attached, and can be resized. Should only be called on flexible dtypes. Types that are attached to an array will always be sized, hence the array form of this macro not existing.

Changed in version 1.18.

For structured datatypes with no fields this function now returns False.

**PyTypeNum_ISUSERDEF** (int *num*)

**PyDataType_ISUSERDEF** (*PyArray_Descr* descr*)

**PyArray_ISUSERDEF** (*PyArrayObject *obj*)
Type represents a user-defined type.

**PyTypeNum_ISEXTENDED** (int *num*)

**PyDataType_ISEXTENDED** (*PyArray_Descr* descr*)

**PyArray_ISEXTENDED** (*PyArrayObject *obj*)
Type is either flexible or user-defined.

**PyTypeNum_ISOBJECT** (int *num*)

**PyDataType_ISOBJECT** (*PyArray_Descr* descr*)

**PyArray_ISOBJECT** (*PyArrayObject *obj*)
Type represents object data type.

**PyTypeNum_ISBOOL** (int *num*)

**PyDataType_ISBOOL** (*PyArray_Descr* descr*)

**PyArray_ISBOOL** (*PyArrayObject *obj*)
Type represents Boolean data type.

**PyDataType_HASFIELDS** (*PyArray_Descr* descr*)

**PyArray_HASFIELDS** (*PyArrayObject *obj*)
Type has fields associated with it.

**PyArray_ISNOTSWAPPED** (m)
Evaluates true if the data area of the ndarray *m* is in machine byte-order according to the array's data-type descriptor.

**PyArray_ISBYTESWAPPED** (m)
Evaluates true if the data area of the ndarray *m* is **not** in machine byte-order according to the array's data-type descriptor.

Bool **PyArray_EquivTypes** (*PyArray_Descr* type1*, *PyArray_Descr* type2*)
Return `NPY_TRUE` if *type1* and *type2* actually represent equivalent types for this platform (the fortran member of each type is ignored). For example, on 32-bit platforms, `NPY_LONG` and `NPY_INT` are equivalent. Otherwise return `NPY_FALSE`.

Bool **PyArray_EquivArrTypes** (*PyArrayObject* a1*, *PyArrayObject * a2*)
Return `NPY_TRUE` if *a1* and *a2* are arrays with equivalent types for this platform.

Bool **PyArray_EquivTypenums** (int *typenum1*, int *typenum2*)
Special case of `PyArray_EquivTypes` (…) that does not accept flexible data types but may be easier to call.

int **PyArray_EquivByteorders** ({byteorder} *b1*, {byteorder} *b2*)
True if byteorder characters ( `NPY_LITTLE`, `NPY_BIG`, `NPY_NATIVE`, `NPY_IGNORE` ) are either equal or

equivalent as to their specification of a native byte order. Thus, on a little-endian machine `NPY_LITTLE` and `NPY_NATIVE` are equivalent where they are not equivalent on a big-endian machine.

### Converting data types

PyObject* **`PyArray_Cast`** (*PyArrayObject* arr*, int *typenum*)

Mainly for backwards compatibility to the Numeric C-API and for simple casts to non-flexible types. Return a new array object with the elements of *arr* cast to the data-type *typenum* which must be one of the enumerated types and not a flexible type.

PyObject* **`PyArray_CastToType`** (*PyArrayObject* arr*, *PyArray_Descr* type*, int *fortran*)

Return a new array of the *type* specified, casting the elements of *arr* as appropriate. The fortran argument specifies the ordering of the output array.

int **`PyArray_CastTo`** (*PyArrayObject* out*, *PyArrayObject* in*)

As of 1.6, this function simply calls `PyArray_CopyInto`, which handles the casting.

Cast the elements of the array *in* into the array *out*. The output array should be writeable, have an integer-multiple of the number of elements in the input array (more than one copy can be placed in out), and have a data type that is one of the builtin types. Returns 0 on success and -1 if an error occurs.

PyArray_VectorUnaryFunc* **`PyArray_GetCastFunc`** (*PyArray_Descr* from*, int *totype*)

Return the low-level casting function to cast from the given descriptor to the builtin type number. If no casting function exists return `NULL` and set an error. Using this function instead of direct access to *from* ->f->cast will allow support of any user-defined casting functions added to a descriptors casting dictionary.

int **`PyArray_CanCastSafely`** (int *fromtype*, int *totype*)

Returns non-zero if an array of data type *fromtype* can be cast to an array of data type *totype* without losing information. An exception is that 64-bit integers are allowed to be cast to 64-bit floating point values even though this can lose precision on large integers so as not to proliferate the use of long doubles without explicit requests. Flexible array types are not checked according to their lengths with this function.

int **`PyArray_CanCastTo`** (*PyArray_Descr* fromtype*, *PyArray_Descr* totype*)

`PyArray_CanCastTypeTo` supersedes this function in NumPy 1.6 and later.

Equivalent to PyArray_CanCastTypeTo(fromtype, totype, NPY_SAFE_CASTING).

int **`PyArray_CanCastTypeTo`** (*PyArray_Descr* fromtype*, *PyArray_Descr* totype*, *NPY_CASTING casting*)

New in version 1.6.

Returns non-zero if an array of data type *fromtype* (which can include flexible types) can be cast safely to an array of data type *totype* (which can include flexible types) according to the casting rule *casting*. For simple types with `NPY_SAFE_CASTING`, this is basically a wrapper around `PyArray_CanCastSafely`, but for flexible types such as strings or unicode, it produces results taking into account their sizes. Integer and float types can only be cast to a string or unicode type using `NPY_SAFE_CASTING` if the string or unicode type is big enough to hold the max value of the integer/float type being cast from.

int **`PyArray_CanCastArrayTo`** (*PyArrayObject* arr*, *PyArray_Descr* totype*, *NPY_CASTING casting*)

New in version 1.6.

Returns non-zero if *arr* can be cast to *totype* according to the casting rule given in *casting*. If *arr* is an array scalar, its value is taken into account, and non-zero is also returned when the value will not overflow or be truncated to an integer when converting to a smaller type.

This is almost the same as the result of PyArray_CanCastTypeTo(PyArray_MinScalarType(arr), totype, casting), but it also handles a special case arising because the set of uint values is not a subset of the int values for types with the same number of bits.

*PyArray_Descr** **`PyArray_MinScalarType`** (*PyArrayObject* arr*)

New in version 1.6.

If *arr* is an array, returns its data type descriptor, but if *arr* is an array scalar (has 0 dimensions), it finds the data type of smallest size to which the value may be converted without overflow or truncation to an integer.

This function will not demote complex to float or anything to boolean, but will demote a signed integer to an unsigned integer when the scalar value is positive.

*PyArray_Descr** **`PyArray_PromoteTypes`** (*PyArray_Descr** *type1*, *PyArray_Descr** *type2*)

New in version 1.6.

Finds the data type of smallest size and kind to which *type1* and *type2* may be safely converted. This function is symmetric and associative. A string or unicode result will be the proper size for storing the max value of the input types converted to a string or unicode.

*PyArray_Descr** **`PyArray_ResultType`** (*npy_intp* *narrs*, *PyArrayObject***arrs, *npy_intp* *ndtypes*, *PyArray_Descr***dtypes)

New in version 1.6.

This applies type promotion to all the inputs, using the NumPy rules for combining scalars and arrays, to determine the output type of a set of operands. This is the same result type that ufuncs produce. The specific algorithm used is as follows.

Categories are determined by first checking which of boolean, integer (int/uint), or floating point (float/complex) the maximum kind of all the arrays and the scalars are.

If there are only scalars or the maximum category of the scalars is higher than the maximum category of the arrays, the data types are combined with `PyArray_PromoteTypes` to produce the return value.

Otherwise, PyArray_MinScalarType is called on each array, and the resulting data types are all combined with `PyArray_PromoteTypes` to produce the return value.

The set of int values is not a subset of the uint values for types with the same number of bits, something not reflected in `PyArray_MinScalarType`, but handled as a special case in PyArray_ResultType.

int **`PyArray_ObjectType`** (PyObject* *op*, int *mintype*)

This function is superseded by `PyArray_MinScalarType` and/or `PyArray_ResultType`.

This function is useful for determining a common type that two or more arrays can be converted to. It only works for non-flexible array types as no itemsize information is passed. The *mintype* argument represents the minimum type acceptable, and *op* represents the object that will be converted to an array. The return value is the enumerated typenumber that represents the data-type that *op* should have.

void **`PyArray_ArrayType`** (PyObject* *op*, *PyArray_Descr** *mintype*, *PyArray_Descr** *outtype*)

This function is superseded by `PyArray_ResultType`.

This function works similarly to `PyArray_ObjectType` (…) except it handles flexible arrays. The *mintype* argument can have an itemsize member and the *outtype* argument will have an itemsize member at least as big but perhaps bigger depending on the object *op*.

*PyArrayObject*** **`PyArray_ConvertToCommonType`** (PyObject* *op*, int* *n*)

The functionality this provides is largely superseded by iterator `NpyIter` introduced in 1.6, with flag `NPY_ITER_COMMON_DTYPE` or with the same dtype parameter for all operands.

Convert a sequence of Python objects contained in *op* to an array of ndarrays each having the same data type. The type is selected in the same way as *PyArray_ResultType*. The length of the sequence is returned in *n*, and an *n* -length array of `PyArrayObject` pointers is the return value (or `NULL` if an error occurs). The returned array must be freed by the caller of this routine (using `PyDataMem_FREE` ) and all the array objects in it `DECREF` ‘d or a memory-leak will occur. The example template-code below shows a typically usage:

Changed in version 1.18.0: A mix of scalars and zero-dimensional arrays now produces a type capable of holding the scalar value. Previously priority was given to the dtype of the arrays.

```
mps = PyArray_ConvertToCommonType(obj, &n);
if (mps==NULL) return NULL;
{code}
<before return>
for (i=0; i<n; i++) Py_DECREF(mps[i]);
PyDataMem_FREE(mps);
{return}
```

char* **`PyArray_Zero`**(*PyArrayObject* arr*)

A pointer to newly created memory of size *arr* ->itemsize that holds the representation of 0 for that type. The returned pointer, *ret*, **must be freed** using *`PyDataMem_FREE`* (ret) when it is not needed anymore.

char* **`PyArray_One`**(*PyArrayObject* arr*)

A pointer to newly created memory of size *arr* ->itemsize that holds the representation of 1 for that type. The returned pointer, *ret*, **must be freed** using *`PyDataMem_FREE`* (ret) when it is not needed anymore.

int **`PyArray_ValidType`**(int *typenum*)

Returns *`NPY_TRUE`* if *typenum* represents a valid type-number (builtin or user-defined or character code). Otherwise, this function returns *`NPY_FALSE`*.

### New data types

void **`PyArray_InitArrFuncs`**(*PyArray_ArrFuncs* f*)

Initialize all function pointers and members to `NULL`.

int **`PyArray_RegisterDataType`**(*PyArray_Descr* dtype*)

Register a data-type as a new user-defined data type for arrays. The type must have most of its entries filled in. This is not always checked and errors can produce segfaults. In particular, the typeobj member of the `dtype` structure must be filled with a Python type that has a fixed-size element-size that corresponds to the elsize member of *dtype*. Also the `f` member must have the required functions: nonzero, copyswap, copyswapn, getitem, setitem, and cast (some of the cast functions may be `NULL` if no support is desired). To avoid confusion, you should choose a unique character typecode but this is not enforced and not relied on internally.

A user-defined type number is returned that uniquely identifies the type. A pointer to the new structure can then be obtained from *`PyArray_DescrFromType`* using the returned type number. A -1 is returned if an error occurs. If this *dtype* has already been registered (checked only by the address of the pointer), then return the previously-assigned type-number.

int **`PyArray_RegisterCastFunc`**(*PyArray_Descr* descr*, int *totype*, PyArray_VectorUnaryFunc* *castfunc*)

Register a low-level casting function, *castfunc*, to convert from the data-type, *descr*, to the given data-type number, *totype*. Any old casting function is over-written. A `0` is returned on success or a `-1` on failure.

int **`PyArray_RegisterCanCast`**(*PyArray_Descr* descr*, int *totype*, *NPY_SCALARKIND scalar*)

Register the data-type number, *totype*, as castable from data-type object, *descr*, of the given *scalar* kind. Use *scalar* = *`NPY_NOSCALAR`* to register that an array of data-type *descr* can be cast safely to a data-type whose type_number is *totype*.

### Special functions for NPY_OBJECT

int **PyArray_INCREF** (*PyArrayObject* op*)

Used for an array, *op*, that contains any Python objects. It increments the reference count of every object in the array according to the data-type of *op*. A -1 is returned if an error occurs, otherwise 0 is returned.

void **PyArray_Item_INCREF** (char* *ptr*, *PyArray_Descr* dtype*)

A function to INCREF all the objects at the location *ptr* according to the data-type *dtype*. If *ptr* is the start of a structured type with an object at any offset, then this will (recursively) increment the reference count of all object-like items in the structured type.

int **PyArray_XDECREF** (*PyArrayObject* op*)

Used for an array, *op*, that contains any Python objects. It decrements the reference count of every object in the array according to the data-type of *op*. Normal return value is 0. A -1 is returned if an error occurs.

void **PyArray_Item_XDECREF** (char* *ptr*, *PyArray_Descr* dtype*)

A function to XDECREF all the object-like items at the location *ptr* as recorded in the data-type, *dtype*. This works recursively so that if `dtype` itself has fields with data-types that contain object-like items, all the object-like fields will be XDECREF `'d`.

void **PyArray_FillObjectArray** (*PyArrayObject* arr*, PyObject* *obj*)

Fill a newly created array with a single value obj at all locations in the structure with object data-types. No checking is performed but *arr* must be of data-type `NPY_OBJECT` and be single-segment and uninitialized (no previous objects in position). Use `PyArray_DECREF` (*arr*) if you need to decrement all the items in the object array prior to calling this function.

int **PyArray_SetUpdateIfCopyBase** (*PyArrayObject* arr*, *PyArrayObject* base*)

Precondition: `arr` is a copy of `base` (though possibly with different strides, ordering, etc.) Set the UPDATEIFCOPY flag and `arr->base` so that when `arr` is destructed, it will copy any changes back to `base`. DEPRECATED, use `PyArray_SetWritebackIfCopyBase``.

Returns 0 for success, -1 for failure.

int **PyArray_SetWritebackIfCopyBase** (*PyArrayObject* arr*, *PyArrayObject* base*)

Precondition: `arr` is a copy of `base` (though possibly with different strides, ordering, etc.) Sets the `NPY_ARRAY_WRITEBACKIFCOPY` flag and `arr->base`, and set `base` to READONLY. Call `PyArray_ResolveWritebackIfCopy` before calling *Py_DECREF‘* in order copy any changes back to `base` and reset the READONLY flag.

Returns 0 for success, -1 for failure.

## 8.4.4 Array flags

The `flags` attribute of the `PyArrayObject` structure contains important information about the memory used by the array (pointed to by the data member) This flag information must be kept accurate or strange results and even segfaults may result.

There are 6 (binary) flags that describe the memory area used by the data buffer. These constants are defined in `arrayobject.h` and determine the bit-position of the flag. Python exposes a nice attribute- based interface as well as a dictionary-like interface for getting (and, if appropriate, setting) these flags.

Memory areas of all kinds can be pointed to by an ndarray, necessitating these flags. If you get an arbitrary `PyArrayObject` in C-code, you need to be aware of the flags that are set. If you need to guarantee a certain kind of array (like `NPY_ARRAY_C_CONTIGUOUS` and `NPY_ARRAY_BEHAVED`), then pass these requirements into the PyArray_FromAny function.

### Basic Array Flags

An ndarray can have a data segment that is not a simple contiguous chunk of well-behaved memory you can manipulate. It may not be aligned with word boundaries (very important on some platforms). It might have its data in a different byte-order than the machine recognizes. It might not be writeable. It might be in Fortran-contiguous order. The array flags are used to indicate what can be said about data associated with an array.

In versions 1.6 and earlier of NumPy, the following flags did not have the _ARRAY_ macro namespace in them. That form of the constant names is deprecated in 1.7.

**NPY_ARRAY_C_CONTIGUOUS**
The data area is in C-style contiguous order (last index varies the fastest).

**NPY_ARRAY_F_CONTIGUOUS**
The data area is in Fortran-style contiguous order (first index varies the fastest).

---

**Note:** Arrays can be both C-style and Fortran-style contiguous simultaneously. This is clear for 1-dimensional arrays, but can also be true for higher dimensional arrays.

Even for contiguous arrays a stride for a given dimension `arr.strides[dim]` may be *arbitrary* if `arr.shape[dim] == 1` or the array has no elements. It does *not* generally hold that `self.strides[-1] == self.itemsize` for C-style contiguous arrays or `self.strides[0] == self.itemsize` for Fortran-style contiguous arrays is true. The correct way to access the `itemsize` of an array from the C API is `PyArray_ITEMSIZE(arr)`.

**See also:**

*Internal memory layout of an ndarray*

---

**NPY_ARRAY_OWNDATA**
The data area is owned by this array.

**NPY_ARRAY_ALIGNED**
The data area and all array elements are aligned appropriately.

**NPY_ARRAY_WRITEABLE**
The data area can be written to.

Notice that the above 3 flags are defined so that a new, well- behaved array has these flags defined as true.

**NPY_ARRAY_WRITEBACKIFCOPY**
The data area represents a (well-behaved) copy whose information should be transferred back to the original when *`PyArray_ResolveWritebackIfCopy`* is called.

This is a special flag that is set if this array represents a copy made because a user required certain flags in *`PyArray_FromAny`* and a copy had to be made of some other array (and the user asked for this flag to be set in such a situation). The base attribute then points to the "misbehaved" array (which is set read_only). :c:func'PyArray_ResolveWritebackIfCopy' will copy its contents back to the "misbehaved" array (casting if necessary) and will reset the "misbehaved" array to *`NPY_ARRAY_WRITEABLE`*. If the "misbehaved" array was not *`NPY_ARRAY_WRITEABLE`* to begin with then *`PyArray_FromAny`* would have returned an error because *`NPY_ARRAY_WRITEBACKIFCOPY`* would not have been possible.

**NPY_ARRAY_UPDATEIFCOPY**
A deprecated version of *`NPY_ARRAY_WRITEBACKIFCOPY`* which depends upon `dealloc` to trigger the writeback. For backwards compatibility, *`PyArray_ResolveWritebackIfCopy`* is called at `dealloc` but relying on that behavior is deprecated and not supported in PyPy.

*`PyArray_UpdateFlags`* (obj, flags) will update the `obj->flags` for `flags` which can be any of *`NPY_ARRAY_C_CONTIGUOUS`*, *`NPY_ARRAY_F_CONTIGUOUS`*, *`NPY_ARRAY_ALIGNED`*, or

*NPY_ARRAY_WRITEABLE*.

### Combinations of array flags

**NPY_ARRAY_BEHAVED**
*NPY_ARRAY_ALIGNED* | *NPY_ARRAY_WRITEABLE*

**NPY_ARRAY_CARRAY**
*NPY_ARRAY_C_CONTIGUOUS* | *NPY_ARRAY_BEHAVED*

**NPY_ARRAY_CARRAY_RO**
*NPY_ARRAY_C_CONTIGUOUS* | *NPY_ARRAY_ALIGNED*

**NPY_ARRAY_FARRAY**
*NPY_ARRAY_F_CONTIGUOUS* | *NPY_ARRAY_BEHAVED*

**NPY_ARRAY_FARRAY_RO**
*NPY_ARRAY_F_CONTIGUOUS* | *NPY_ARRAY_ALIGNED*

**NPY_ARRAY_DEFAULT**
*NPY_ARRAY_CARRAY*

**NPY_ARRAY_UPDATE_ALL**
*NPY_ARRAY_C_CONTIGUOUS* | *NPY_ARRAY_F_CONTIGUOUS* | *NPY_ARRAY_ALIGNED*

### Flag-like constants

These constants are used in *PyArray_FromAny* (and its macro forms) to specify desired properties of the new array.

**NPY_ARRAY_FORCECAST**
Cast to the desired type, even if it can't be done without losing information.

**NPY_ARRAY_ENSURECOPY**
Make sure the resulting array is a copy of the original.

**NPY_ARRAY_ENSUREARRAY**
Make sure the resulting object is an actual ndarray, and not a sub-class.

**NPY_ARRAY_NOTSWAPPED**
Only used in *PyArray_CheckFromAny* to over-ride the byteorder of the data-type object passed in.

**NPY_ARRAY_BEHAVED_NS**
*NPY_ARRAY_ALIGNED* | *NPY_ARRAY_WRITEABLE* | *NPY_ARRAY_NOTSWAPPED*

### Flag checking

For all of these macros *arr* must be an instance of a (subclass of) *PyArray_Type*.

**PyArray_CHKFLAGS** (PyObject **arr*, flags)
The first parameter, arr, must be an ndarray or subclass. The parameter, *flags*, should be an integer consisting of bitwise combinations of the possible flags an array can have: *NPY_ARRAY_C_CONTIGUOUS*, *NPY_ARRAY_F_CONTIGUOUS*, *NPY_ARRAY_OWNDATA*, *NPY_ARRAY_ALIGNED*, *NPY_ARRAY_WRITEABLE*, *NPY_ARRAY_WRITEBACKIFCOPY*, *NPY_ARRAY_UPDATEIFCOPY*.

**PyArray_IS_C_CONTIGUOUS** (PyObject **arr*)
Evaluates true if *arr* is C-style contiguous.

**PyArray_IS_F_CONTIGUOUS**(PyObject **arr*)
Evaluates true if *arr* is Fortran-style contiguous.

**PyArray_ISFORTRAN**(PyObject **arr*)
Evaluates true if *arr* is Fortran-style contiguous and *not* C-style contiguous. *PyArray_IS_F_CONTIGUOUS* is the correct way to test for Fortran-style contiguity.

**PyArray_ISWRITEABLE**(PyObject **arr*)
Evaluates true if the data area of *arr* can be written to

**PyArray_ISALIGNED**(PyObject **arr*)
Evaluates true if the data area of *arr* is properly aligned on the machine.

**PyArray_ISBEHAVED**(PyObject **arr*)
Evaluates true if the data area of *arr* is aligned and writeable and in machine byte-order according to its descriptor.

**PyArray_ISBEHAVED_RO**(PyObject **arr*)
Evaluates true if the data area of *arr* is aligned and in machine byte-order.

**PyArray_ISCARRAY**(PyObject **arr*)
Evaluates true if the data area of *arr* is C-style contiguous, and *PyArray_ISBEHAVED* (*arr*) is true.

**PyArray_ISFARRAY**(PyObject **arr*)
Evaluates true if the data area of *arr* is Fortran-style contiguous and *PyArray_ISBEHAVED* (*arr*) is true.

**PyArray_ISCARRAY_RO**(PyObject **arr*)
Evaluates true if the data area of *arr* is C-style contiguous, aligned, and in machine byte-order.

**PyArray_ISFARRAY_RO**(PyObject **arr*)
Evaluates true if the data area of *arr* is Fortran-style contiguous, aligned, and in machine byte-order **.**

**PyArray_ISONESEGMENT**(PyObject **arr*)
Evaluates true if the data area of *arr* consists of a single (C-style or Fortran-style) contiguous segment.

void **PyArray_UpdateFlags**(*PyArrayObject** *arr*, int *flagmask*)
The *NPY_ARRAY_C_CONTIGUOUS*, *NPY_ARRAY_ALIGNED*, and *NPY_ARRAY_F_CONTIGUOUS* array flags can be "calculated" from the array object itself. This routine updates one or more of these flags of *arr* as specified in *flagmask* by performing the required calculation.

**Warning:** It is important to keep the flags updated (using *PyArray_UpdateFlags* can help) whenever a manipulation with an array is performed that might cause them to change. Later calculations in NumPy that rely on the state of these flags do not repeat the calculation to update them.

### 8.4.5 Array method alternative API

#### Conversion

PyObject* **PyArray_GetField**(*PyArrayObject** *self*, *PyArray_Descr** *dtype*, int *offset*)
Equivalent to *ndarray.getfield* (*self*, *dtype*, *offset*). This function steals a reference to *PyArray_Descr* and returns a new array of the given *dtype* using the data in the current array at a specified *offset* in bytes. The *offset* plus the itemsize of the new array type must be less than `self ->descr->elsize` or an error is raised. The same shape and strides as the original array are used. Therefore, this function has the effect of returning a field from a structured array. But, it can also be used to select specific bytes or groups of bytes from any array type.

int **PyArray_SetField**(*PyArrayObject** *self*, *PyArray_Descr** *dtype*, int *offset*, PyObject* *val*)
Equivalent to *ndarray.setfield* (*self*, *val*, *dtype*, *offset* ). Set the field starting at *offset* in bytes and of the given *dtype* to *val*. The *offset* plus *dtype* ->elsize must be less than *self* ->descr->elsize or an error is raised.

Otherwise, the *val* argument is converted to an array and copied into the field pointed to. If necessary, the elements of *val* are repeated to fill the destination array, But, the number of elements in the destination must be an integer multiple of the number of elements in *val*.

PyObject* **`PyArray_Byteswap`**(*PyArrayObject** *self*, Bool *inplace*)

Equivalent to `ndarray.byteswap` (*self*, *inplace*). Return an array whose data area is byteswapped. If *inplace* is non-zero, then do the byteswap inplace and return a reference to self. Otherwise, create a byteswapped copy and leave self unchanged.

PyObject* **`PyArray_NewCopy`**(*PyArrayObject** *old*, *NPY_ORDER* *order*)

Equivalent to `ndarray.copy` (*self*, *fortran*). Make a copy of the *old* array. The returned array is always aligned and writeable with data interpreted the same as the old array. If *order* is `NPY_CORDER`, then a C-style contiguous array is returned. If *order* is `NPY_FORTRANORDER`, then a Fortran-style contiguous array is returned. If *order is* `NPY_ANYORDER`, then the array returned is Fortran-style contiguous only if the old one is; otherwise, it is C-style contiguous.

PyObject* **`PyArray_ToList`**(*PyArrayObject** *self*)

Equivalent to `ndarray.tolist` (*self*). Return a nested Python list from *self*.

PyObject* **`PyArray_ToString`**(*PyArrayObject** *self*, *NPY_ORDER* *order*)

Equivalent to `ndarray.tobytes` (*self*, *order*). Return the bytes of this array in a Python string.

PyObject* **`PyArray_ToFile`**(*PyArrayObject** *self*, FILE* *fp*, char* *sep*, char* *format*)

Write the contents of *self* to the file pointer *fp* in C-style contiguous fashion. Write the data as binary bytes if *sep* is the string ""or `NULL`. Otherwise, write the contents of *self* as text using the *sep* string as the item separator. Each item will be printed to the file. If the *format* string is not `NULL` or "", then it is a Python print statement format string showing how the items are to be written.

int **`PyArray_Dump`**(PyObject* *self*, PyObject* *file*, int *protocol*)

Pickle the object in *self* to the given *file* (either a string or a Python file object). If *file* is a Python string it is considered to be the name of a file which is then opened in binary mode. The given *protocol* is used (if *protocol* is negative, or the highest available is used). This is a simple wrapper around cPickle.dump(*self*, *file*, *protocol*).

PyObject* **`PyArray_Dumps`**(PyObject* *self*, int *protocol*)

Pickle the object in *self* to a Python string and return it. Use the Pickle *protocol* provided (or the highest available if *protocol* is negative).

int **`PyArray_FillWithScalar`**(*PyArrayObject** *arr*, PyObject* *obj*)

Fill the array, *arr*, with the given scalar object, *obj*. The object is first converted to the data type of *arr*, and then copied into every location. A -1 is returned if an error occurs, otherwise 0 is returned.

PyObject* **`PyArray_View`**(*PyArrayObject** *self*, *PyArray_Descr** *dtype*, PyTypeObject **ptype*)

Equivalent to `ndarray.view` (*self*, *dtype*). Return a new view of the array *self* as possibly a different data-type, *dtype*, and different array subclass *ptype*.

If *dtype* is `NULL`, then the returned array will have the same data type as *self*. The new data-type must be consistent with the size of *self*. Either the itemsizes must be identical, or *self* must be single-segment and the total number of bytes must be the same. In the latter case the dimensions of the returned array will be altered in the last (or first for Fortran-style contiguous arrays) dimension. The data area of the returned array and self is exactly the same.

### Shape Manipulation

PyObject* **PyArray_Newshape** (*PyArrayObject* * *self*, *PyArray_Dims* * *newshape*, *NPY_ORDER order*)

Result will be a new array (pointing to the same memory location as *self* if possible), but having a shape given by *newshape*. If the new shape is not compatible with the strides of *self*, then a copy of the array with the new specified shape will be returned.

PyObject* **PyArray_Reshape** (*PyArrayObject* * *self*, PyObject* *shape*)

Equivalent to `ndarray.reshape` (*self*, *shape*) where *shape* is a sequence. Converts *shape* to a `PyArray_Dims` structure and calls `PyArray_Newshape` internally. For back-ward compatibility – Not recommended

PyObject* **PyArray_Squeeze** (*PyArrayObject* * *self*)

Equivalent to `ndarray.squeeze` (*self*). Return a new view of *self* with all of the dimensions of length 1 removed from the shape.

> **Warning:** matrix objects are always 2-dimensional. Therefore, `PyArray_Squeeze` has no effect on arrays of matrix sub-class.

PyObject* **PyArray_SwapAxes** (*PyArrayObject* * *self*, int *a1*, int *a2*)

Equivalent to `ndarray.swapaxes` (*self*, *a1*, *a2*). The returned array is a new view of the data in *self* with the given axes, *a1* and *a2*, swapped.

PyObject* **PyArray_Resize** (*PyArrayObject* * *self*, *PyArray_Dims* * *newshape*, int *refcheck*, *NPY_ORDER fortran*)

Equivalent to `ndarray.resize` (*self*, *newshape*, refcheck = *refcheck*, order= fortran ). This function only works on single-segment arrays. It changes the shape of *self* inplace and will reallocate the memory for *self* if *newshape* has a different total number of elements then the old shape. If reallocation is necessary, then *self* must own its data, have *self* - `>base==NULL`, have *self* - `>weakrefs==NULL`, and (unless refcheck is 0) not be referenced by any other array. The fortran argument can be `NPY_ANYORDER`, `NPY_CORDER`, or `NPY_FORTRANORDER`. It currently has no effect. Eventually it could be used to determine how the resize operation should view the data when constructing a differently-dimensioned array. Returns None on success and NULL on error.

PyObject* **PyArray_Transpose** (*PyArrayObject* * *self*, *PyArray_Dims* * *permute*)

Equivalent to `ndarray.transpose` (*self*, *permute*). Permute the axes of the ndarray object *self* according to the data structure *permute* and return the result. If *permute* is `NULL`, then the resulting array has its axes reversed. For example if *self* has shape $10 \times 20 \times 30$, and *permute* `.ptr` is (0,2,1) the shape of the result is $10 \times 30 \times 20$. If *permute* is `NULL`, the shape of the result is $30 \times 20 \times 10$.

PyObject* **PyArray_Flatten** (*PyArrayObject* * *self*, *NPY_ORDER order*)

Equivalent to `ndarray.flatten` (*self*, *order*). Return a 1-d copy of the array. If *order* is `NPY_FORTRANORDER` the elements are scanned out in Fortran order (first-dimension varies the fastest). If *order* is `NPY_CORDER`, the elements of `self` are scanned in C-order (last dimension varies the fastest). If *order* `NPY_ANYORDER`, then the result of `PyArray_ISFORTRAN` (*self*) is used to determine which order to flatten.

PyObject* **PyArray_Ravel** (*PyArrayObject* * *self*, *NPY_ORDER order*)

Equivalent to *self*.ravel(*order*). Same basic functionality as `PyArray_Flatten` (*self*, *order*) except if *order* is 0 and *self* is C-style contiguous, the shape is altered but no copy is performed.

### Item selection and manipulation

PyObject* **PyArray_TakeFrom**(*PyArrayObject* self*, PyObject* *indices*, int *axis*, *PyArrayObject* ret*, *NPY_CLIPMODE clipmode*)

Equivalent to `ndarray.take` (*self*, *indices*, *axis*, *ret*, *clipmode*) except *axis* =None in Python is obtained by setting *axis* = `NPY_MAXDIMS` in C. Extract the items from self indicated by the integer-valued *indices* along the given *axis*. The clipmode argument can be `NPY_RAISE`, `NPY_WRAP`, or `NPY_CLIP` to indicate what to do with out-of-bound indices. The *ret* argument can specify an output array rather than having one created internally.

PyObject* **PyArray_PutTo**(*PyArrayObject* self*, PyObject* *values*, PyObject* *indices*, *NPY_CLIPMODE clipmode*)

Equivalent to *self*.put(*values*, *indices*, *clipmode* ). Put *values* into *self* at the corresponding (flattened) *indices*. If *values* is too small it will be repeated as necessary.

PyObject* **PyArray_PutMask**(*PyArrayObject* self*, PyObject* *values*, PyObject* *mask*)

Place the *values* in *self* wherever corresponding positions (using a flattened context) in *mask* are true. The *mask* and *self* arrays must have the same total number of elements. If *values* is too small, it will be repeated as necessary.

PyObject* **PyArray_Repeat**(*PyArrayObject* self*, PyObject* *op*, int *axis*)

Equivalent to `ndarray.repeat` (*self*, *op*, *axis*). Copy the elements of *self*, *op* times along the given *axis*. Either *op* is a scalar integer or a sequence of length *self* ->dimensions[ *axis* ] indicating how many times to repeat each item along the axis.

PyObject* **PyArray_Choose**(*PyArrayObject* self*, PyObject* *op*, *PyArrayObject* ret*, *NPY_CLIPMODE clipmode*)

Equivalent to `ndarray.choose` (*self*, *op*, *ret*, *clipmode*). Create a new array by selecting elements from the sequence of arrays in *op* based on the integer values in *self*. The arrays must all be broadcastable to the same shape and the entries in *self* should be between 0 and len(*op*). The output is placed in *ret* unless it is `NULL` in which case a new output is created. The *clipmode* argument determines behavior for when entries in *self* are not between 0 and len(*op*).

**NPY_RAISE**

raise a ValueError;

**NPY_WRAP**

wrap values < 0 by adding len(*op*) and values >=len(*op*) by subtracting len(*op*) until they are in range;

**NPY_CLIP**

all values are clipped to the region [0, len(*op*) ).

PyObject* **PyArray_Sort**(*PyArrayObject* self*, int *axis*, *NPY_SORTKIND kind*)

Equivalent to `ndarray.sort` (*self*, *axis*, *kind*). Return an array with the items of *self* sorted along *axis*. The array is sorted using the algorithm denoted by *kind*, which is an integer/enum pointing to the type of sorting algorithms used.

PyObject* **PyArray_ArgSort**(*PyArrayObject* self*, int *axis*)

Equivalent to `ndarray.argsort` (*self*, *axis*). Return an array of indices such that selection of these indices along the given `axis` would return a sorted version of *self*. If *self* ->descr is a data-type with fields defined, then self->descr->names is used to determine the sort order. A comparison where the first field is equal will use the second field and so on. To alter the sort order of a structured array, create a new data-type with a different order of names and construct a view of the array with that new data-type.

PyObject* **PyArray_LexSort**(PyObject* *sort_keys*, int *axis*)

Given a sequence of arrays (*sort_keys*) of the same shape, return an array of indices (similar to `PyArray_ArgSort` (...)) that would sort the arrays lexicographically. A lexicographic sort specifies that when two keys are found to be equal, the order is based on comparison of subsequent keys. A merge sort (which leaves equal entries unmoved) is required to be defined for the types. The sort is accomplished by sorting the indices first using the first *sort_key* and then using the second *sort_key* and so forth. This is equivalent to the lexsort(*sort_keys*, *axis*) Python command. Because of the way the merge-sort works, be sure to understand the order the *sort_keys* must be in (reversed from the order you would use when comparing two elements).

If these arrays are all collected in a structured array, then `PyArray_Sort` (…) can also be used to sort the array directly.

PyObject* **`PyArray_SearchSorted`** (*PyArrayObject* self*, PyObject* *values*, NPY_SEARCHSIDE *side*, PyObject* *perm*)

Equivalent to `ndarray.searchsorted` (*self*, *values*, *side*, *perm*). Assuming *self* is a 1-d array in ascending order, then the output is an array of indices the same shape as *values* such that, if the elements in *values* were inserted before the indices, the order of *self* would be preserved. No checking is done on whether or not self is in ascending order.

The *side* argument indicates whether the index returned should be that of the first suitable location (if `NPY_SEARCHLEFT`) or of the last (if `NPY_SEARCHRIGHT`).

The *sorter* argument, if not `NULL`, must be a 1D array of integer indices the same length as *self*, that sorts it into ascending order. This is typically the result of a call to `PyArray_ArgSort` (…) Binary search is used to find the required insertion points.

int **`PyArray_Partition`** (*PyArrayObject *self*, *PyArrayObject * ktharray*, int *axis*, NPY_SELECTKIND *which*)

Equivalent to `ndarray.partition` (*self*, *ktharray*, *axis*, *kind*). Partitions the array so that the values of the element indexed by *ktharray* are in the positions they would be if the array is fully sorted and places all elements smaller than the kth before and all elements equal or greater after the kth element. The ordering of all elements within the partitions is undefined. If *self*->descr is a data-type with fields defined, then self->descr->names is used to determine the sort order. A comparison where the first field is equal will use the second field and so on. To alter the sort order of a structured array, create a new data-type with a different order of names and construct a view of the array with that new data-type. Returns zero on success and -1 on failure.

PyObject* **`PyArray_ArgPartition`** (*PyArrayObject *op*, *PyArrayObject * ktharray*, int *axis*, NPY_SELECTKIND *which*)

Equivalent to `ndarray.argpartition` (*self*, *ktharray*, *axis*, *kind*). Return an array of indices such that selection of these indices along the given `axis` would return a partitioned version of *self*.

PyObject* **`PyArray_Diagonal`** (*PyArrayObject* self*, int *offset*, int *axis1*, int *axis2*)

Equivalent to `ndarray.diagonal` (*self*, *offset*, *axis1*, *axis2* ). Return the *offset* diagonals of the 2-d arrays defined by *axis1* and *axis2*.

*npy_intp* **`PyArray_CountNonzero`** (*PyArrayObject* self*)

New in version 1.6.

Counts the number of non-zero elements in the array object *self*.

PyObject* **`PyArray_Nonzero`** (*PyArrayObject* self*)

Equivalent to `ndarray.nonzero` (*self*). Returns a tuple of index arrays that select elements of *self* that are nonzero. If (nd= `PyArray_NDIM` ( `self` ))==1, then a single index array is returned. The index arrays have data type `NPY_INTP`. If a tuple is returned (nd $\neq$ 1), then its length is nd.

PyObject* **`PyArray_Compress`** (*PyArrayObject* self*, PyObject* *condition*, int *axis*, *PyArrayObject* out*)

Equivalent to `ndarray.compress` (*self*, *condition*, *axis* ). Return the elements along *axis* corresponding to elements of *condition* that are true.

### Calculation

---

**Tip:** Pass in `NPY_MAXDIMS` for axis in order to achieve the same effect that is obtained by passing in `axis=None` in Python (treating the array as a 1-d array).

---

**Note:** The out argument specifies where to place the result. If out is NULL, then the output array is created, otherwise the output is placed in out which must be the correct size and type. A new reference to the output array is always returned even when out is not NULL. The caller of the routine has the responsibility to `Py_DECREF` out if not NULL or a memory-leak will occur.

---

PyObject* **`PyArray_ArgMax`** (*PyArrayObject* self*, int *axis*, *PyArrayObject* out*)

Equivalent to `ndarray.argmax` (*self*, *axis*). Return the index of the largest element of *self* along *axis*.

PyObject* **`PyArray_ArgMin`** (*PyArrayObject* self*, int *axis*, *PyArrayObject* out*)

Equivalent to `ndarray.argmin` (*self*, *axis*). Return the index of the smallest element of *self* along *axis*.

PyObject* **`PyArray_Max`** (*PyArrayObject* self*, int *axis*, *PyArrayObject* out*)

Equivalent to `ndarray.max` (*self*, *axis*). Returns the largest element of *self* along the given *axis*. When the result is a single element, returns a numpy scalar instead of an ndarray.

PyObject* **`PyArray_Min`** (*PyArrayObject* self*, int *axis*, *PyArrayObject* out*)

Equivalent to `ndarray.min` (*self*, *axis*). Return the smallest element of *self* along the given *axis*. When the result is a single element, returns a numpy scalar instead of an ndarray.

PyObject* **`PyArray_Ptp`** (*PyArrayObject* self*, int *axis*, *PyArrayObject* out*)

Equivalent to `ndarray.ptp` (*self*, *axis*). Return the difference between the largest element of *self* along *axis* and the smallest element of *self* along *axis*. When the result is a single element, returns a numpy scalar instead of an ndarray.

---

**Note:** The rtype argument specifies the data-type the reduction should take place over. This is important if the data-type of the array is not "large" enough to handle the output. By default, all integer data-types are made at least as large as `NPY_LONG` for the "add" and "multiply" ufuncs (which form the basis for mean, sum, cumsum, prod, and cumprod functions).

---

PyObject* **`PyArray_Mean`** (*PyArrayObject* self*, int *axis*, int *rtype*, *PyArrayObject* out*)

Equivalent to `ndarray.mean` (*self*, *axis*, *rtype*). Returns the mean of the elements along the given *axis*, using the enumerated type *rtype* as the data type to sum in. Default sum behavior is obtained using `NPY_NOTYPE` for *rtype*.

PyObject* **`PyArray_Trace`** (*PyArrayObject* self*, int *offset*, int *axis1*, int *axis2*, int *rtype*, *PyArrayObject* out*)

Equivalent to `ndarray.trace` (*self*, *offset*, *axis1*, *axis2*, *rtype*). Return the sum (using *rtype* as the data type of summation) over the *offset* diagonal elements of the 2-d arrays defined by *axis1* and *axis2* variables. A positive offset chooses diagonals above the main diagonal. A negative offset selects diagonals below the main diagonal.

PyObject* **`PyArray_Clip`** (*PyArrayObject* self*, PyObject* *min*, PyObject* *max*)

Equivalent to `ndarray.clip` (*self*, *min*, *max*). Clip an array, *self*, so that values larger than *max* are fixed to *max* and values less than *min* are fixed to *min*.

PyObject* **`PyArray_Conjugate`** (*PyArrayObject* self*)

Equivalent to `ndarray.conjugate` (*self*). Return the complex conjugate of *self*. If *self* is not of complex data type, then return *self* with a reference.

PyObject* **`PyArray_Round`** (*PyArrayObject* self*, int *decimals*, *PyArrayObject* out*)

Equivalent to `ndarray.round` (*self*, *decimals*, *out*). Returns the array with elements rounded to the nearest decimal place. The decimal place is defined as the $10^{-\text{decimals}}$ digit so that negative *decimals* cause rounding to the nearest 10's, 100's, etc. If out is `NULL`, then the output array is created, otherwise the output is placed in *out* which must be the correct size and type.

PyObject* **`PyArray_Std`** (*PyArrayObject* * *self*, int *axis*, int *rtype*, *PyArrayObject* * *out*)

Equivalent to `ndarray.std` (*self*, *axis*, *rtype*). Return the standard deviation using data along *axis* converted to data type *rtype*.

PyObject* **`PyArray_Sum`** (*PyArrayObject* * *self*, int *axis*, int *rtype*, *PyArrayObject* * *out*)

Equivalent to `ndarray.sum` (*self*, *axis*, *rtype*). Return 1-d vector sums of elements in *self* along *axis*. Perform the sum after converting data to data type *rtype*.

PyObject* **`PyArray_CumSum`** (*PyArrayObject* * *self*, int *axis*, int *rtype*, *PyArrayObject* * *out*)

Equivalent to `ndarray.cumsum` (*self*, *axis*, *rtype*). Return cumulative 1-d sums of elements in *self* along *axis*. Perform the sum after converting data to data type *rtype*.

PyObject* **`PyArray_Prod`** (*PyArrayObject* * *self*, int *axis*, int *rtype*, *PyArrayObject* * *out*)

Equivalent to `ndarray.prod` (*self*, *axis*, *rtype*). Return 1-d products of elements in *self* along *axis*. Perform the product after converting data to data type *rtype*.

PyObject* **`PyArray_CumProd`** (*PyArrayObject* * *self*, int *axis*, int *rtype*, *PyArrayObject* * *out*)

Equivalent to `ndarray.cumprod` (*self*, *axis*, *rtype*). Return 1-d cumulative products of elements in `self` along `axis`. Perform the product after converting data to data type `rtype`.

PyObject* **`PyArray_All`** (*PyArrayObject* * *self*, int *axis*, *PyArrayObject* * *out*)

Equivalent to `ndarray.all` (*self*, *axis*). Return an array with True elements for every 1-d sub-array of `self` defined by `axis` in which all the elements are True.

PyObject* **`PyArray_Any`** (*PyArrayObject* * *self*, int *axis*, *PyArrayObject* * *out*)

Equivalent to `ndarray.any` (*self*, *axis*). Return an array with True elements for every 1-d sub-array of *self* defined by *axis* in which any of the elements are True.

## 8.4.6 Functions

### Array Functions

int **`PyArray_AsCArray`** (PyObject** *op*, void* *ptr*, *npy_intp* * *dims*, int *nd*, int *typenum*, int *itemsize*)

Sometimes it is useful to access a multidimensional array as a C-style multi-dimensional array so that algorithms can be implemented using C's a[i][j][k] syntax. This routine returns a pointer, *ptr*, that simulates this kind of C-style array, for 1-, 2-, and 3-d ndarrays.

**Parameters**

- **`op`** – The address to any Python object. This Python object will be replaced with an equivalent well-behaved, C-style contiguous, ndarray of the given data type specified by the last two arguments. Be sure that stealing a reference in this way to the input object is justified.
- **`ptr`** – The address to a (ctype* for 1-d, ctype** for 2-d or ctype*** for 3-d) variable where ctype is the equivalent C-type for the data type. On return, *ptr* will be addressable as a 1-d, 2-d, or 3-d array.
- **`dims`** – An output array that contains the shape of the array object. This array gives boundaries on any looping that will take place.
- **`nd`** – The dimensionality of the array (1, 2, or 3).
- **`typenum`** – The expected data type of the array.

- **`itemsize`** – This argument is only needed when *typenum* represents a flexible array. Otherwise it should be 0.

---

**Note:** The simulation of a C-style array is not complete for 2-d and 3-d arrays. For example, the simulated arrays of pointers cannot be passed to subroutines expecting specific, statically-defined 2-d and 3-d arrays. To pass to functions requiring those kind of inputs, you must statically define the required array and copy data.

---

int **`PyArray_Free`** (PyObject* *op*, void* *ptr*)

Must be called with the same objects and memory locations returned from `PyArray_AsCArray` (…). This function cleans up memory that otherwise would get leaked.

PyObject* **`PyArray_Concatenate`** (PyObject* *obj*, int *axis*)

Join the sequence of objects in *obj* together along *axis* into a single array. If the dimensions or types are not compatible an error is raised.

PyObject* **`PyArray_InnerProduct`** (PyObject* *obj1*, PyObject* *obj2*)

Compute a product-sum over the last dimensions of *obj1* and *obj2*. Neither array is conjugated.

PyObject* **`PyArray_MatrixProduct`** (PyObject* *obj1*, PyObject* *obj*)

Compute a product-sum over the last dimension of *obj1* and the second-to-last dimension of *obj2*. For 2-d arrays this is a matrix-product. Neither array is conjugated.

PyObject* **`PyArray_MatrixProduct2`** (PyObject* *obj1*, PyObject* *obj*, *PyArrayObject* out*)

New in version 1.6.

Same as PyArray_MatrixProduct, but store the result in *out*. The output array must have the correct shape, type, and be C-contiguous, or an exception is raised.

PyObject* **`PyArray_EinsteinSum`** (char* *subscripts*, *npy_intp nop*, *PyArrayObject** op_in*, *PyArray_Descr* dtype*, *NPY_ORDER order*, *NPY_CASTING casting*, *PyArrayObject* out*)

New in version 1.6.

Applies the Einstein summation convention to the array operands provided, returning a new array or placing the result in *out*. The string in *subscripts* is a comma separated list of index letters. The number of operands is in *nop*, and *op_in* is an array containing those operands. The data type of the output can be forced with *dtype*, the output order can be forced with *order* (`NPY_KEEPORDER` is recommended), and when *dtype* is specified, *casting* indicates how permissive the data conversion should be.

See the `einsum` function for more details.

PyObject* **`PyArray_CopyAndTranspose`** (PyObject * *op*)

A specialized copy and transpose function that works only for 2-d arrays. The returned array is a transposed copy of *op*.

PyObject* **`PyArray_Correlate`** (PyObject* *op1*, PyObject* *op2*, int *mode*)

Compute the 1-d correlation of the 1-d arrays *op1* and *op2* . The correlation is computed at each output point by multiplying *op1* by a shifted version of *op2* and summing the result. As a result of the shift, needed values outside of the defined range of *op1* and *op2* are interpreted as zero. The mode determines how many shifts to return: 0 - return only shifts that did not need to assume zero- values; 1 - return an object that is the same size as *op1*, 2 - return all possible shifts (any overlap at all is accepted).

### Notes

This does not compute the usual correlation: if op2 is larger than op1, the arguments are swapped, and the conjugate is never taken for complex arrays. See PyArray_Correlate2 for the usual signal processing correlation.

PyObject* **`PyArray_Correlate2`** (PyObject* *op1*, PyObject* *op2*, int *mode*)

Updated version of PyArray_Correlate, which uses the usual definition of correlation for 1d arrays. The correlation is computed at each output point by multiplying *op1* by a shifted version of *op2* and summing the result. As a result of the shift, needed values outside of the defined range of *op1* and *op2* are interpreted as zero. The mode determines how many shifts to return: 0 - return only shifts that did not need to assume zero- values; 1 - return an object that is the same size as *op1*, 2 - return all possible shifts (any overlap at all is accepted).

### Notes

Compute z as follows:

```
z[k] = sum_n op1[n] * conj(op2[n+k])
```

PyObject* **`PyArray_Where`** (PyObject* *condition*, PyObject* *x*, PyObject* *y*)

If both `x` and `y` are `NULL`, then return *`PyArray_Nonzero`* (*condition*). Otherwise, both *x* and *y* must be given and the object returned is shaped like *condition* and has elements of *x* and *y* where *condition* is respectively True or False.

## Other functions

Bool **`PyArray_CheckStrides`** (int *elsize*, int *nd*, *npy_intp* *numbytes*, *npy_intp* const* *dims*, *npy_intp* const* *newstrides*)

Determine if *newstrides* is a strides array consistent with the memory of an *nd* -dimensional array with shape `dims` and element-size, *elsize*. The *newstrides* array is checked to see if jumping by the provided number of bytes in each direction will ever mean jumping more than *numbytes* which is the assumed size of the available memory segment. If *numbytes* is 0, then an equivalent *numbytes* is computed assuming *nd*, *dims*, and *elsize* refer to a single-segment array. Return *`NPY_TRUE`* if *newstrides* is acceptable, otherwise return *`NPY_FALSE`*.

*npy_intp* **`PyArray_MultiplyList`** (*npy_intp* const* *seq*, int *n*)

int **`PyArray_MultiplyIntList`** (int const* *seq*, int *n*)

Both of these routines multiply an *n* -length array, *seq*, of integers and return the result. No overflow checking is performed.

int **`PyArray_CompareLists`** (*npy_intp* const* *l1*, *npy_intp* const* *l2*, int *n*)

Given two *n* -length arrays of integers, *l1*, and *l2*, return 1 if the lists are identical; otherwise, return 0.

## 8.4.7 Auxiliary Data With Object Semantics

New in version 1.7.0.

**`NpyAuxData`**

When working with more complex dtypes which are composed of other dtypes, such as the struct dtype, creating inner loops that manipulate the dtypes requires carrying along additional data. NumPy supports this idea through a struct *`NpyAuxData`*, mandating a few conventions so that it is possible to do this.

Defining an *`NpyAuxData`* is similar to defining a class in C++, but the object semantics have to be tracked manually since the API is in C. Here's an example for a function which doubles up an element using an element copier function as a primitive.:

```
typedef struct {
    NpyAuxData base;
    ElementCopier_Func *func;
    NpyAuxData *funcdata;
} eldoubler_aux_data;

void free_element_doubler_aux_data(NpyAuxData *data)
{
    eldoubler_aux_data *d = (eldoubler_aux_data *)data;
    /* Free the memory owned by this auxdata */
    NPY_AUXDATA_FREE(d->funcdata);
    PyArray_free(d);
}

NpyAuxData *clone_element_doubler_aux_data(NpyAuxData *data)
{
    eldoubler_aux_data *ret = PyArray_malloc(sizeof(eldoubler_aux_data));
    if (ret == NULL) {
        return NULL;
    }

    /* Raw copy of all data */
    memcpy(ret, data, sizeof(eldoubler_aux_data));

    /* Fix up the owned auxdata so we have our own copy */
    ret->funcdata = NPY_AUXDATA_CLONE(ret->funcdata);
    if (ret->funcdata == NULL) {
        PyArray_free(ret);
        return NULL;
    }

    return (NpyAuxData *)ret;
}

NpyAuxData *create_element_doubler_aux_data(
                            ElementCopier_Func *func,
                            NpyAuxData *funcdata)
{
    eldoubler_aux_data *ret = PyArray_malloc(sizeof(eldoubler_aux_data));
    if (ret == NULL) {
        PyErr_NoMemory();
        return NULL;
    }
    memset(&ret, 0, sizeof(eldoubler_aux_data));
    ret->base->free = &free_element_doubler_aux_data;
    ret->base->clone = &clone_element_doubler_aux_data;
    ret->func = func;
    ret->funcdata = funcdata;

    return (NpyAuxData *)ret;
}
```

**`NpyAuxData_FreeFunc`**

The function pointer type for NpyAuxData free functions.

**`NpyAuxData_CloneFunc`**

The function pointer type for NpyAuxData clone functions. These functions should never set the Python exception on error, because they may be called from a multi-threaded context.

**NPY_AUXDATA_FREE** (auxdata)
A macro which calls the auxdata's free function appropriately, does nothing if auxdata is NULL.

**NPY_AUXDATA_CLONE** (auxdata)
A macro which calls the auxdata's clone function appropriately, returning a deep copy of the auxiliary data.

## 8.4.8 Array Iterators

As of NumPy 1.6.0, these array iterators are superseded by the new array iterator, *NpyIter*.

An array iterator is a simple way to access the elements of an N-dimensional array quickly and efficiently. Section *2* provides more description and examples of this useful approach to looping over an array.

PyObject* **PyArray_IterNew** (PyObject* *arr*)
Return an array iterator object from the array, *arr*. This is equivalent to *arr*. **flat**. The array iterator object makes it easy to loop over an N-dimensional non-contiguous array in C-style contiguous fashion.

PyObject* **PyArray_IterAllButAxis** (PyObject* *arr*, int **axis*)
Return an array iterator that will iterate over all axes but the one provided in **axis*. The returned iterator cannot be used with *PyArray_ITER_GOTO1D*. This iterator could be used to write something similar to what ufuncs do wherein the loop over the largest axis is done by a separate sub-routine. If **axis* is negative then **axis* will be set to the axis having the smallest stride and that axis will be used.

PyObject ***PyArray_BroadcastToShape** (PyObject* *arr*, *npy_intp* const **dimensions*, int *nd*)
Return an array iterator that is broadcast to iterate as an array of the shape provided by *dimensions* and *nd*.

int **PyArrayIter_Check** (PyObject* *op*)
Evaluates true if *op* is an array iterator (or instance of a subclass of the array iterator type).

void **PyArray_ITER_RESET** (PyObject* *iterator*)
Reset an *iterator* to the beginning of the array.

void **PyArray_ITER_NEXT** (PyObject* *iterator*)
Incremement the index and the dataptr members of the *iterator* to point to the next element of the array. If the array is not (C-style) contiguous, also increment the N-dimensional coordinates array.

void ***PyArray_ITER_DATA** (PyObject* *iterator*)
A pointer to the current element of the array.

void **PyArray_ITER_GOTO** (PyObject* *iterator*, *npy_intp** *destination*)
Set the *iterator* index, dataptr, and coordinates members to the location in the array indicated by the N-dimensional c-array, *destination*, which must have size at least *iterator* ->nd_m1+1.

**PyArray_ITER_GOTO1D** (PyObject* *iterator*, *npy_intp* *index*)
Set the *iterator* index and dataptr to the location in the array indicated by the integer *index* which points to an element in the C-styled flattened array.

int **PyArray_ITER_NOTDONE** (PyObject* *iterator*)
Evaluates TRUE as long as the iterator has not looped through all of the elements, otherwise it evaluates FALSE.

### 8.4.9 Broadcasting (multi-iterators)

PyObject* **PyArray_MultiIterNew**(int *num*, ...)

A simplified interface to broadcasting. This function takes the number of arrays to broadcast and then *num* extra ( `PyObject *` ) arguments. These arguments are converted to arrays and iterators are created. `PyArray_Broadcast` is then called on the resulting multi-iterator object. The resulting, broadcasted mult-iterator object is then returned. A broadcasted operation can then be performed using a single loop and using `PyArray_MultiIter_NEXT` (..)

void **PyArray_MultiIter_RESET**(PyObject* *multi*)

Reset all the iterators to the beginning in a multi-iterator object, *multi*.

void **PyArray_MultiIter_NEXT**(PyObject* *multi*)

Advance each iterator in a multi-iterator object, *multi*, to its next (broadcasted) element.

void ***PyArray_MultiIter_DATA**(PyObject* *multi*, int *i*)

Return the data-pointer of the $i^{\text{th}}$ iterator in a multi-iterator object.

void **PyArray_MultiIter_NEXTi**(PyObject* *multi*, int *i*)

Advance the pointer of only the $i^{\text{th}}$ iterator.

void **PyArray_MultiIter_GOTO**(PyObject* *multi*, *npy_intp** *destination*)

Advance each iterator in a multi-iterator object, *multi*, to the given $N$ -dimensional *destination* where $N$ is the number of dimensions in the broadcasted array.

void **PyArray_MultiIter_GOTO1D**(PyObject* *multi*, *npy_intp* *index*)

Advance each iterator in a multi-iterator object, *multi*, to the corresponding location of the *index* into the flattened broadcasted array.

int **PyArray_MultiIter_NOTDONE**(PyObject* *multi*)

Evaluates TRUE as long as the multi-iterator has not looped through all of the elements (of the broadcasted result), otherwise it evaluates FALSE.

int **PyArray_Broadcast**(*PyArrayMultiIterObject** *mit*)

This function encapsulates the broadcasting rules. The *mit* container should already contain iterators for all the arrays that need to be broadcast. On return, these iterators will be adjusted so that iteration over each simultaneously will accomplish the broadcasting. A negative number is returned if an error occurs.

int **PyArray_RemoveSmallest**(*PyArrayMultiIterObject** *mit*)

This function takes a multi-iterator object that has been previously "broadcasted," finds the dimension with the smallest "sum of strides" in the broadcasted result and adapts all the iterators so as not to iterate over that dimension (by effectively making them of length-1 in that dimension). The corresponding dimension is returned unless *mit* ->nd is 0, then -1 is returned. This function is useful for constructing ufunc-like routines that broadcast their inputs correctly and then call a strided 1-d version of the routine as the inner-loop. This 1-d version is usually optimized for speed and for this reason the loop should be performed over the axis that won't require large stride jumps.

### 8.4.10 Neighborhood iterator

New in version 1.4.0.

Neighborhood iterators are subclasses of the iterator object, and can be used to iter over a neighborhood of a point. For example, you may want to iterate over every voxel of a 3d image, and for every such voxel, iterate over an hypercube. Neighborhood iterator automatically handle boundaries, thus making this kind of code much easier to write than manual boundaries handling, at the cost of a slight overhead.

PyObject* **PyArray_NeighborhoodIterNew**(*PyArrayIterObject** *iter*, *npy_intp* *bounds*, int *mode*, *PyArrayObject** *fill_value*)

This function creates a new neighborhood iterator from an existing iterator. The neighborhood will be computed relatively to the position currently pointed by *iter*, the bounds define the shape of the neighborhood iterator, and the mode argument the boundaries handling mode.

The *bounds* argument is expected to be a (2 * iter->ao->nd) arrays, such as the range bound[2*i]->bounds[2*i+1] defines the range where to walk for dimension i (both bounds are included in the walked coordinates). The bounds should be ordered for each dimension (bounds[2*i] <= bounds[2*i+1]).

The mode should be one of:

**`NPY_NEIGHBORHOOD_ITER_ZERO_PADDING`**
Zero padding. Outside bounds values will be 0.

**`NPY_NEIGHBORHOOD_ITER_ONE_PADDING`**
One padding, Outside bounds values will be 1.

**`NPY_NEIGHBORHOOD_ITER_CONSTANT_PADDING`**
Constant padding. Outside bounds values will be the same as the first item in fill_value.

**`NPY_NEIGHBORHOOD_ITER_MIRROR_PADDING`**
Mirror padding. Outside bounds values will be as if the array items were mirrored. For example, for the array [1, 2, 3, 4], x[-2] will be 2, x[-2] will be 1, x[4] will be 4, x[5] will be 1, etc…

**`NPY_NEIGHBORHOOD_ITER_CIRCULAR_PADDING`**
Circular padding. Outside bounds values will be as if the array was repeated. For example, for the array [1, 2, 3, 4], x[-2] will be 3, x[-2] will be 4, x[4] will be 1, x[5] will be 2, etc…

If the mode is constant filling (*NPY_NEIGHBORHOOD_ITER_CONSTANT_PADDING*), fill_value should point to an array object which holds the filling value (the first item will be the filling value if the array contains more than one item). For other cases, fill_value may be NULL.

- The iterator holds a reference to iter
- Return NULL on failure (in which case the reference count of iter is not changed)
- iter itself can be a Neighborhood iterator: this can be useful for .e.g automatic boundaries handling
- the object returned by this function should be safe to use as a normal iterator
- If the position of iter is changed, any subsequent call to PyArrayNeighborhoodIter_Next is undefined behavior, and PyArrayNeighborhoodIter_Reset must be called.

```
PyArrayIterObject *iter;
PyArrayNeighborhoodIterObject *neigh_iter;
iter = PyArray_IterNew(x);

/*For a 3x3 kernel */
bounds = {-1, 1, -1, 1};
neigh_iter = (PyArrayNeighborhoodIterObject*)PyArrayNeighborhoodIter_New(
     iter, bounds, NPY_NEIGHBORHOOD_ITER_ZERO_PADDING, NULL);

for(i = 0; i < iter->size; ++i) {
     for (j = 0; j < neigh_iter->size; ++j) {
             /* Walk around the item currently pointed by iter->dataptr */
             PyArrayNeighborhoodIter_Next(neigh_iter);
     }

     /* Move to the next point of iter */
     PyArrayIter_Next(iter);
     PyArrayNeighborhoodIter_Reset(neigh_iter);
}
```

int **PyArrayNeighborhoodIter_Reset** (*PyArrayNeighborhoodIterObject* iter*)

Reset the iterator position to the first point of the neighborhood. This should be called whenever the iter argument given at PyArray_NeighborhoodIterObject is changed (see example)

int **PyArrayNeighborhoodIter_Next** (*PyArrayNeighborhoodIterObject* iter*)

After this call, iter->dataptr points to the next point of the neighborhood. Calling this function after every point of the neighborhood has been visited is undefined.

## 8.4.11 Array Scalars

PyObject* **PyArray_Return** (*PyArrayObject* arr*)

This function steals a reference to *arr*.

This function checks to see if *arr* is a 0-dimensional array and, if so, returns the appropriate array scalar. It should be used whenever 0-dimensional arrays could be returned to Python.

PyObject* **PyArray_Scalar** (void* *data*, *PyArray_Descr* dtype*, PyObject* *itemsize*)

Return an array scalar object of the given enumerated *typenum* and *itemsize* by **copying** from memory pointed to by *data* . If *swap* is nonzero then this function will byteswap the data if appropriate to the data-type because array scalars are always in correct machine-byte order.

PyObject* **PyArray_ToScalar** (void* *data*, *PyArrayObject* arr*)

Return an array scalar object of the type and itemsize indicated by the array object *arr* copied from the memory pointed to by *data* and swapping if the data in *arr* is not in machine byte-order.

PyObject* **PyArray_FromScalar** (PyObject* *scalar*, *PyArray_Descr* outcode*)

Return a 0-dimensional array of type determined by *outcode* from *scalar* which should be an array-scalar object. If *outcode* is NULL, then the type is determined from *scalar*.

void **PyArray_ScalarAsCtype** (PyObject* *scalar*, void* *ctypeptr*)

Return in *ctypeptr* a pointer to the actual value in an array scalar. There is no error checking so *scalar* must be an array-scalar object, and ctypeptr must have enough space to hold the correct type. For flexible-sized types, a pointer to the data is copied into the memory of *ctypeptr*, for all other types, the actual data is copied into the address pointed to by *ctypeptr*.

void **PyArray_CastScalarToCtype** (PyObject* *scalar*, void* *ctypeptr*, *PyArray_Descr* outcode*)

Return the data (cast to the data type indicated by *outcode*) from the array-scalar, *scalar*, into the memory pointed to by *ctypeptr* (which must be large enough to handle the incoming memory).

PyObject* **PyArray_TypeObjectFromType** (int *type*)

Returns a scalar type-object from a type-number, *type* . Equivalent to `PyArray_DescrFromType` (*type*)->typeobj except for reference counting and error-checking. Returns a new reference to the typeobject on success or `NULL` on failure.

*NPY_SCALARKIND* **PyArray_ScalarKind** (int *typenum*, *PyArrayObject** arr*)

See the function `PyArray_MinScalarType` for an alternative mechanism introduced in NumPy 1.6.0.

Return the kind of scalar represented by *typenum* and the array in **arr* (if *arr* is not `NULL` ). The array is assumed to be rank-0 and only used if *typenum* represents a signed integer. If *arr* is not `NULL` and the first element is negative then `NPY_INTNEG_SCALAR` is returned, otherwise `NPY_INTPOS_SCALAR` is returned. The possible return values are the enumerated values in `NPY_SCALARKIND`.

int **PyArray_CanCoerceScalar** (char *thistype*, char *neededtype*, *NPY_SCALARKIND scalar*)

See the function `PyArray_ResultType` for details of NumPy type promotion, updated in NumPy 1.6.0.

Implements the rules for scalar coercion. Scalars are only silently coerced from thistype to neededtype if this function returns nonzero. If scalar is `NPY_NOSCALAR`, then this function is equivalent to `PyArray_CanCastSafely`. The rule is that scalars of the same KIND can be coerced into arrays of the same KIND. This rule means that high-precision scalars will never cause low-precision arrays of the same KIND to be upcast.

### 8.4.12 Data-type descriptors

**Warning:** Data-type objects must be reference counted so be aware of the action on the data-type reference of different C-API calls. The standard rule is that when a data-type object is returned it is a new reference. Functions that take `PyArray_Descr *` objects and return arrays steal references to the data-type their inputs unless otherwise noted. Therefore, you must own a reference to any data-type object used as input to such a function.

int **`PyArray_DescrCheck`** (PyObject* *obj*)

Evaluates as true if *obj* is a data-type object ( `PyArray_Descr *` ).

*PyArray_Descr** **`PyArray_DescrNew`** (*PyArray_Descr** *obj*)

Return a new data-type object copied from *obj* (the fields reference is just updated so that the new object points to the same fields dictionary if any).

*PyArray_Descr** **`PyArray_DescrNewFromType`** (int *typenum*)

Create a new data-type object from the built-in (or user-registered) data-type indicated by *typenum*. All builtin types should not have any of their fields changed. This creates a new copy of the `PyArray_Descr` structure so that you can fill it in as appropriate. This function is especially needed for flexible data-types which need to have a new elsize member in order to be meaningful in array construction.

*PyArray_Descr** **`PyArray_DescrNewByteorder`** (*PyArray_Descr** *obj*, char *newendian*)

Create a new data-type object with the byteorder set according to *newendian*. All referenced data-type objects (in subdescr and fields members of the data-type object) are also changed (recursively). If a byteorder of `NPY_IGNORE` is encountered it is left alone. If newendian is `NPY_SWAP`, then all byte-orders are swapped. Other valid newendian values are `NPY_NATIVE`, `NPY_LITTLE`, and `NPY_BIG` which all cause the returned data-typed descriptor (and all it's referenced data-type descriptors) to have the corresponding byte- order.

*PyArray_Descr** **`PyArray_DescrFromObject`** (PyObject* *op*, *PyArray_Descr** *mintype*)

Determine an appropriate data-type object from the object *op* (which should be a "nested" sequence object) and the minimum data-type descriptor mintype (which can be `NULL` ). Similar in behavior to array(*op*).dtype. Don't confuse this function with `PyArray_DescrConverter`. This function essentially looks at all the objects in the (nested) sequence and determines the data-type from the elements it finds.

*PyArray_Descr** **`PyArray_DescrFromScalar`** (PyObject* *scalar*)

Return a data-type object from an array-scalar object. No checking is done to be sure that *scalar* is an array scalar. If no suitable data-type can be determined, then a data-type of `NPY_OBJECT` is returned by default.

*PyArray_Descr** **`PyArray_DescrFromType`** (int *typenum*)

Returns a data-type object corresponding to *typenum*. The *typenum* can be one of the enumerated types, a character code for one of the enumerated types, or a user-defined type. If you want to use a flexible size array, then you need to `flexible typenum` and set the results `elsize` parameter to the desired size. The typenum is one of the `NPY_TYPES`.

int **`PyArray_DescrConverter`** (PyObject* *obj*, *PyArray_Descr*** *dtype*)

Convert any compatible Python object, *obj*, to a data-type object in *dtype*. A large number of Python objects can be converted to data-type objects. See *Data type objects (dtype)* for a complete description. This version of the converter converts None objects to a `NPY_DEFAULT_TYPE` data-type object. This function can be used with the "O&" character code in `PyArg_ParseTuple` processing.

int **`PyArray_DescrConverter2`** (PyObject* *obj*, *PyArray_Descr*** *dtype*)

Convert any compatible Python object, *obj*, to a data-type object in *dtype*. This version of the converter converts

None objects so that the returned data-type is `NULL`. This function can also be used with the "O&" character in PyArg_ParseTuple processing.

int **`Pyarray_DescrAlignConverter`**(PyObject* *obj*, *PyArray_Descr*** *dtype*)
Like *`PyArray_DescrConverter`* except it aligns C-struct-like objects on word-boundaries as the compiler would.

int **`Pyarray_DescrAlignConverter2`**(PyObject* *obj*, *PyArray_Descr*** *dtype*)
Like *`PyArray_DescrConverter2`* except it aligns C-struct-like objects on word-boundaries as the compiler would.

PyObject ***`PyArray_FieldNames`**(PyObject* *dict*)
Take the fields dictionary, *dict*, such as the one attached to a data-type object and construct an ordered-list of field names such as is stored in the names field of the *`PyArray_Descr`* object.

## 8.4.13 Conversion Utilities

### For use with `PyArg_ParseTuple`

All of these functions can be used in `PyArg_ParseTuple`(…) with the "O&" format specifier to automatically convert any Python object to the required C-object. All of these functions return *`NPY_SUCCEED`* if successful and *`NPY_FAIL`* if not. The first argument to all of these function is a Python object. The second argument is the **address** of the C-type to convert the Python object to.

**Warning:** Be sure to understand what steps you should take to manage the memory when using these conversion functions. These functions can require freeing memory, and/or altering the reference counts of specific objects based on your use.

int **`PyArray_Converter`**(PyObject* *obj*, PyObject** *address*)
Convert any Python object to a *`PyArrayObject`*. If *`PyArray_Check`* (*obj*) is TRUE then its reference count is incremented and a reference placed in *address*. If *obj* is not an array, then convert it to an array using *`PyArray_FromAny`* . No matter what is returned, you must DECREF the object returned by this routine in *address* when you are done with it.

int **`PyArray_OutputConverter`**(PyObject* *obj*, *PyArrayObject*** *address*)
This is a default converter for output arrays given to functions. If *obj* is `Py_None` or `NULL`, then **address* will be `NULL` but the call will succeed. If *`PyArray_Check`* ( *obj*) is TRUE then it is returned in **address* without incrementing its reference count.

int **`PyArray_IntpConverter`**(PyObject* *obj*, *PyArray_Dims** *seq*)
Convert any Python sequence, *obj*, smaller than *`NPY_MAXDIMS`* to a C-array of *`npy_intp`*. The Python object could also be a single number. The *seq* variable is a pointer to a structure with members ptr and len. On successful return, *seq* ->ptr contains a pointer to memory that must be freed, by calling *`PyDimMem_FREE`*, to avoid a memory leak. The restriction on memory size allows this converter to be conveniently used for sequences intended to be interpreted as array shapes.

int **`PyArray_BufferConverter`**(PyObject* *obj*, *PyArray_Chunk** *buf*)
Convert any Python object, *obj*, with a (single-segment) buffer interface to a variable with members that detail the object's use of its chunk of memory. The *buf* variable is a pointer to a structure with base, ptr, len, and flags members. The *`PyArray_Chunk`* structure is binary compatible with the Python's buffer object (through its len member on 32-bit platforms and its ptr member on 64-bit platforms or in Python 2.5). On return, the base member is set to *obj* (or its base if *obj* is already a buffer object pointing to another object). If you need to hold on to the memory be sure to INCREF the base member. The chunk of memory is pointed to by *buf* ->ptr member and has

length *buf* ->len. The flags member of *buf* is `NPY_BEHAVED_RO` with the *`NPY_ARRAY_WRITEABLE`* flag set if *obj* has a writeable buffer interface.

int **`PyArray_AxisConverter`** (PyObject * *obj*, int* *axis*)
Convert a Python object, *obj*, representing an axis argument to the proper value for passing to the functions that take an integer axis. Specifically, if *obj* is None, *axis* is set to *`NPY_MAXDIMS`* which is interpreted correctly by the C-API functions that take axis arguments.

int **`PyArray_BoolConverter`** (PyObject* *obj*, Bool* *value*)
Convert any Python object, *obj*, to *`NPY_TRUE`* or *`NPY_FALSE`*, and place the result in *value*.

int **`PyArray_ByteorderConverter`** (PyObject* *obj*, char* *endian*)
Convert Python strings into the corresponding byte-order character: '>', '<', 's', '=', or '|'.

int **`PyArray_SortkindConverter`** (PyObject* *obj*, *NPY_SORTKIND** *sort*)
Convert Python strings into one of *`NPY_QUICKSORT`* (starts with 'q' or 'Q'), *`NPY_HEAPSORT`* (starts with 'h' or 'H'), *`NPY_MERGESORT`* (starts with 'm' or 'M') or *`NPY_STABLESORT`* (starts with 't' or 'T'). *`NPY_MERGESORT`* and *`NPY_STABLESORT`* are aliased to each other for backwards compatibility and may refer to one of several stable sorting algorithms depending on the data type.

int **`PyArray_SearchsideConverter`** (PyObject* *obj*, NPY_SEARCHSIDE* *side*)
Convert Python strings into one of `NPY_SEARCHLEFT` (starts with 'l' or 'L'), or `NPY_SEARCHRIGHT` (starts with 'r' or 'R').

int **`PyArray_OrderConverter`** (PyObject* *obj*, *NPY_ORDER** *order*)
Convert the Python strings 'C', 'F', 'A', and 'K' into the *`NPY_ORDER`* enumeration *`NPY_CORDER`*, *`NPY_FORTRANORDER`*, *`NPY_ANYORDER`*, and *`NPY_KEEPORDER`*.

int **`PyArray_CastingConverter`** (PyObject* *obj*, *NPY_CASTING** *casting*)
Convert the Python strings 'no', 'equiv', 'safe', 'same_kind', and 'unsafe' into the *`NPY_CASTING`* enumeration *`NPY_NO_CASTING`*, *`NPY_EQUIV_CASTING`*, *`NPY_SAFE_CASTING`*, *`NPY_SAME_KIND_CASTING`*, and *`NPY_UNSAFE_CASTING`*.

int **`PyArray_ClipmodeConverter`** (PyObject* *object*, *NPY_CLIPMODE** *val*)
Convert the Python strings 'clip', 'wrap', and 'raise' into the *`NPY_CLIPMODE`* enumeration *`NPY_CLIP`*, *`NPY_WRAP`*, and *`NPY_RAISE`*.

int **`PyArray_ConvertClipmodeSequence`** (PyObject* *object*, *NPY_CLIPMODE** *modes*, int *n*)
Converts either a sequence of clipmodes or a single clipmode into a C array of *`NPY_CLIPMODE`* values. The number of clipmodes *n* must be known before calling this function. This function is provided to help functions allow a different clipmode for each dimension.

### Other conversions

int **`PyArray_PyIntAsInt`** (PyObject* *op*)
Convert all kinds of Python objects (including arrays and array scalars) to a standard integer. On error, -1 is returned and an exception set. You may find useful the macro:

```
#define error_converting(x) (((x) == -1) && PyErr_Occurred())
```

*npy_intp* **`PyArray_PyIntAsIntp`** (PyObject* *op*)
Convert all kinds of Python objects (including arrays and array scalars) to a (platform-pointer-sized) integer. On error, -1 is returned and an exception set.

int **`PyArray_IntpFromSequence`** (PyObject* *seq*, *npy_intp** *vals*, int *maxvals*)
Convert any Python sequence (or single Python number) passed in as *seq* to (up to) *maxvals* pointer-sized integers and place them in the *vals* array. The sequence can be smaller then *maxvals* as the number of converted objects is returned.

int **PyArray_TypestrConvert**(int *itemsize*, int *gentype*)

Convert typestring characters (with *itemsize*) to basic enumerated data types. The typestring character corresponding to signed and unsigned integers, floating point numbers, and complex-floating point numbers are recognized and converted. Other values of gentype are returned. This function can be used to convert, for example, the string 'f4' to `NPY_FLOAT32`.

## 8.4.14 Miscellaneous

### Importing the API

In order to make use of the C-API from another extension module, the `import_array` function must be called. If the extension module is self-contained in a single .c file, then that is all that needs to be done. If, however, the extension module involves multiple files where the C-API is needed then some additional steps must be taken.

void **import_array**(void)

This function must be called in the initialization section of a module that will make use of the C-API. It imports the module where the function-pointer table is stored and points the correct variable to it.

**PY_ARRAY_UNIQUE_SYMBOL**

**NO_IMPORT_ARRAY**

Using these #defines you can use the C-API in multiple files for a single extension module. In each file you must define `PY_ARRAY_UNIQUE_SYMBOL` to some name that will hold the C-API (*e.g.* myextension_ARRAY_API). This must be done **before** including the numpy/arrayobject.h file. In the module initialization routine you call `import_array`. In addition, in the files that do not have the module initialization sub_routine define `NO_IMPORT_ARRAY` prior to including numpy/arrayobject.h.

Suppose I have two files coolmodule.c and coolhelper.c which need to be compiled and linked into a single extension module. Suppose coolmodule.c contains the required initcool module initialization function (with the import_array() function called). Then, coolmodule.c would have at the top:

```
#define PY_ARRAY_UNIQUE_SYMBOL cool_ARRAY_API
#include numpy/arrayobject.h
```

On the other hand, coolhelper.c would contain at the top:

```
#define NO_IMPORT_ARRAY
#define PY_ARRAY_UNIQUE_SYMBOL cool_ARRAY_API
#include numpy/arrayobject.h
```

You can also put the common two last lines into an extension-local header file as long as you make sure that NO_IMPORT_ARRAY is #defined before #including that file.

Internally, these #defines work as follows:

- If neither is defined, the C-API is declared to be `static void**`, so it is only visible within the compilation unit that #includes numpy/arrayobject.h.
- If `PY_ARRAY_UNIQUE_SYMBOL` is #defined, but `NO_IMPORT_ARRAY` is not, the C-API is declared to be `void**`, so that it will also be visible to other compilation units.
- If `NO_IMPORT_ARRAY` is #defined, regardless of whether `PY_ARRAY_UNIQUE_SYMBOL` is, the C-API is declared to be `extern void**`, so it is expected to be defined in another compilation unit.
- Whenever `PY_ARRAY_UNIQUE_SYMBOL` is #defined, it also changes the name of the variable holding the C-API, which defaults to `PyArray_API`, to whatever the macro is #defined to.

### Checking the API Version

Because python extensions are not used in the same way as usual libraries on most platforms, some errors cannot be automatically detected at build time or even runtime. For example, if you build an extension using a function available only for numpy >= 1.3.0, and you import the extension later with numpy 1.2, you will not get an import error (but almost certainly a segmentation fault when calling the function). That's why several functions are provided to check for numpy versions. The macros *`NPY_VERSION`* and `NPY_FEATURE_VERSION` corresponds to the numpy version used to build the extension, whereas the versions returned by the functions PyArray_GetNDArrayCVersion and PyArray_GetNDArrayCFeatureVersion corresponds to the runtime numpy's version.

The rules for ABI and API compatibilities can be summarized as follows:

- Whenever *`NPY_VERSION`* != PyArray_GetNDArrayCVersion, the extension has to be recompiled (ABI incompatibility).
- *`NPY_VERSION`* == PyArray_GetNDArrayCVersion and `NPY_FEATURE_VERSION` <= PyArray_GetNDArrayCFeatureVersion means backward compatible changes.

ABI incompatibility is automatically detected in every numpy's version. API incompatibility detection was added in numpy 1.4.0. If you want to supported many different numpy versions with one extension binary, you have to build your extension with the lowest NPY_FEATURE_VERSION as possible.

unsigned int **`PyArray_GetNDArrayCVersion`** (void)

This just returns the value *`NPY_VERSION`*. *`NPY_VERSION`* changes whenever a backward incompatible change at the ABI level. Because it is in the C-API, however, comparing the output of this function from the value defined in the current header gives a way to test if the C-API has changed thus requiring a re-compilation of extension modules that use the C-API. This is automatically checked in the function *`import_array`*.

unsigned int **`PyArray_GetNDArrayCFeatureVersion`** (void)

New in version 1.4.0.

This just returns the value `NPY_FEATURE_VERSION`. `NPY_FEATURE_VERSION` changes whenever the API changes (e.g. a function is added). A changed value does not always require a recompile.

### Internal Flexibility

int **`PyArray_SetNumericOps`** (PyObject* *dict*)

NumPy stores an internal table of Python callable objects that are used to implement arithmetic operations for arrays as well as certain array calculation methods. This function allows the user to replace any or all of these Python objects with their own versions. The keys of the dictionary, *dict*, are the named functions to replace and the paired value is the Python callable object to use. Care should be taken that the function used to replace an internal array operation does not itself call back to that internal array operation (unless you have designed the function to handle that), or an unchecked infinite recursion can result (possibly causing program crash). The key names that represent operations that can be replaced are:

> **add**, **subtract**, **multiply**, **divide**, **remainder**, **power**, **square**, **reciprocal**, **ones_like**, **sqrt**, **negative**, **positive**, **absolute**, **invert**, **left_shift**, **right_shift**, **bitwise_and**, **bitwise_xor**, **bitwise_or**, **less**, **less_equal**, **equal**, **not_equal**, **greater**, **greater_equal**, **floor_divide**, **true_divide**, **logical_or**, **logical_and**, **floor**, **ceil**, **maximum**, **minimum**, **rint**.

These functions are included here because they are used at least once in the array object's methods. The function returns -1 (without setting a Python Error) if one of the objects being assigned is not callable.

Deprecated since version 1.16.

PyObject* **`PyArray_GetNumericOps`** (void)

Return a Python dictionary containing the callable Python objects stored in the internal arithmetic operation table. The keys of this dictionary are given in the explanation for *`PyArray_SetNumericOps`*.

Deprecated since version 1.16.

void **PyArray_SetStringFunction** (PyObject* *op*, int *repr*)

This function allows you to alter the tp_str and tp_repr methods of the array object to any Python function. Thus you can alter what happens for all arrays when str(arr) or repr(arr) is called from Python. The function to be called is passed in as *op*. If *repr* is non-zero, then this function will be called in response to repr(arr), otherwise the function will be called in response to str(arr). No check on whether or not *op* is callable is performed. The callable passed in to *op* should expect an array argument and should return a string to be printed.

### Memory management

char* **PyDataMem_NEW** (size_t *nbytes*)

**PyDataMem_FREE** (char* *ptr*)

char* **PyDataMem_RENEW** (void * *ptr*, size_t *newbytes*)

Macros to allocate, free, and reallocate memory. These macros are used internally to create arrays.

*npy_intp** **PyDimMem_NEW** (int *nd*)

**PyDimMem_FREE** (char* *ptr*)

*npy_intp** **PyDimMem_RENEW** (void* *ptr*, size_t *newnd*)

Macros to allocate, free, and reallocate dimension and strides memory.

void* **PyArray_malloc** (size_t *nbytes*)

**PyArray_free** (void* *ptr*)

void* **PyArray_realloc** (*npy_intp** *ptr*, size_t *nbytes*)

These macros use different memory allocators, depending on the constant `NPY_USE_PYMEM`. The system malloc is used when `NPY_USE_PYMEM` is 0, if `NPY_USE_PYMEM` is 1, then the Python memory allocator is used.

int **PyArray_ResolveWritebackIfCopy** (*PyArrayObject** *obj*)

If `obj.flags` has *`NPY_ARRAY_WRITEBACKIFCOPY`* or (deprecated) *`NPY_ARRAY_UPDATEIFCOPY`*, this function clears the flags, *DECREF* s *obj->base* and makes it writeable, and sets `obj->base` to NULL. It then copies `obj->data` to *obj->base->data*, and returns the error state of the copy operation. This is the opposite of *`PyArray_SetWritebackIfCopyBase`*. Usually this is called once you are finished with `obj`, just before `Py_DECREF(obj)`. It may be called multiple times, or with `NULL` input. See also *`PyArray_DiscardWritebackIfCopy`*.

Returns 0 if nothing was done, -1 on error, and 1 if action was taken.

### Threading support

These macros are only meaningful if `NPY_ALLOW_THREADS` evaluates True during compilation of the extension module. Otherwise, these macros are equivalent to whitespace. Python uses a single Global Interpreter Lock (GIL) for each Python process so that only a single thread may execute at a time (even on multi-cpu machines). When calling out to a compiled function that may take time to compute (and does not have side-effects for other threads like updated global variables), the GIL should be released so that other Python threads can run while the time-consuming calculations are performed. This can be accomplished using two groups of macros. Typically, if one macro in a group is used in a code block, all of them must be used in the same code block. Currently, `NPY_ALLOW_THREADS` is defined to the python-defined `WITH_THREADS` constant unless the environment variable `NPY_NOSMP` is set in which case `NPY_ALLOW_THREADS` is defined to be 0.

### Group 1

This group is used to call code that may take some time but does not use any Python C-API calls. Thus, the GIL should be released during its calculation.

**NPY_BEGIN_ALLOW_THREADS**
Equivalent to `Py_BEGIN_ALLOW_THREADS` except it uses `NPY_ALLOW_THREADS` to determine if the macro if replaced with white-space or not.

**NPY_END_ALLOW_THREADS**
Equivalent to `Py_END_ALLOW_THREADS` except it uses `NPY_ALLOW_THREADS` to determine if the macro if replaced with white-space or not.

**NPY_BEGIN_THREADS_DEF**
Place in the variable declaration area. This macro sets up the variable needed for storing the Python state.

**NPY_BEGIN_THREADS**
Place right before code that does not need the Python interpreter (no Python C-API calls). This macro saves the Python state and releases the GIL.

**NPY_END_THREADS**
Place right after code that does not need the Python interpreter. This macro acquires the GIL and restores the Python state from the saved variable.

**NPY_BEGIN_THREADS_DESCR**(*PyArray_Descr *dtype*)
Useful to release the GIL only if *dtype* does not contain arbitrary Python objects which may need the Python interpreter during execution of the loop.

**NPY_END_THREADS_DESCR**(*PyArray_Descr *dtype*)
Useful to regain the GIL in situations where it was released using the BEGIN form of this macro.

**NPY_BEGIN_THREADS_THRESHOLDED**(int *loop_size*)
Useful to release the GIL only if *loop_size* exceeds a minimum threshold, currently set to 500. Should be matched with a `NPY_END_THREADS` to regain the GIL.

### Group 2

This group is used to re-acquire the Python GIL after it has been released. For example, suppose the GIL has been released (using the previous calls), and then some path in the code (perhaps in a different subroutine) requires use of the Python C-API, then these macros are useful to acquire the GIL. These macros accomplish essentially a reverse of the previous three (acquire the LOCK saving what state it had) and then re-release it with the saved state.

**NPY_ALLOW_C_API_DEF**
Place in the variable declaration area to set up the necessary variable.

**NPY_ALLOW_C_API**
Place before code that needs to call the Python C-API (when it is known that the GIL has already been released).

**NPY_DISABLE_C_API**
Place after code that needs to call the Python C-API (to re-release the GIL).

**Tip:** Never use semicolons after the threading support macros.

### Priority

**NPY_PRIORITY**
Default priority for arrays.

**NPY_SUBTYPE_PRIORITY**
Default subtype priority.

**NPY_SCALAR_PRIORITY**
Default scalar priority (very small)

double **PyArray_GetPriority** (PyObject* *obj*, double *def* )
Return the `__array_priority__` attribute (converted to a double) of *obj* or *def* if no attribute of that name exists. Fast returns that avoid the attribute lookup are provided for objects of type `PyArray_Type`.

### Default buffers

**NPY_BUFSIZE**
Default size of the user-settable internal buffers.

**NPY_MIN_BUFSIZE**
Smallest size of user-settable internal buffers.

**NPY_MAX_BUFSIZE**
Largest size allowed for the user-settable buffers.

### Other constants

**NPY_NUM_FLOATTYPE**
The number of floating-point types

**NPY_MAXDIMS**
The maximum number of dimensions allowed in arrays.

**NPY_MAXARGS**
The maximum number of array arguments that can be used in functions.

**NPY_VERSION**
The current version of the ndarray object (check to see if this variable is defined to guarantee the numpy/arrayobject.h header is being used).

**NPY_FALSE**
Defined as 0 for use with Bool.

**NPY_TRUE**
Defined as 1 for use with Bool.

**NPY_FAIL**
The return value of failed converter functions which are called using the “O&” syntax in `PyArg_ParseTuple`-like functions.

**NPY_SUCCEED**
The return value of successful converter functions which are called using the “O&” syntax in `PyArg_ParseTuple`-like functions.

### Miscellaneous Macros

**PyArray_SAMESHAPE**(*PyArrayObject *a1*, *PyArrayObject *a2*)
Evaluates as True if arrays *a1* and *a2* have the same shape.

**a**

**b**

**PyArray_MAX**(*a*, *b*)
Returns the maximum of *a* and *b*. If (*a*) or (*b*) are expressions they are evaluated twice.

**PyArray_MIN**(*a*, *b*)
Returns the minimum of *a* and *b*. If (*a*) or (*b*) are expressions they are evaluated twice.

**PyArray_CLT**(*a*, *b*)

**PyArray_CGT**(*a*, *b*)

**PyArray_CLE**(*a*, *b*)

**PyArray_CGE**(*a*, *b*)

**PyArray_CEQ**(*a*, *b*)

**PyArray_CNE**(*a*, *b*)
Implements the complex comparisons between two complex numbers (structures with a real and imag member) using NumPy's definition of the ordering which is lexicographic: comparing the real parts first and then the complex parts if the real parts are equal.

**PyArray_REFCOUNT**(PyObject* *op*)
Returns the reference count of any Python object.

**PyArray_DiscardWritebackIfCopy**(PyObject* *obj*)
If `obj.flags` has *`NPY_ARRAY_WRITEBACKIFCOPY`* or (deprecated) *`NPY_ARRAY_UPDATEIFCOPY`*, this function clears the flags, *DECREF* s *obj->base* and makes it writeable, and sets `obj->base` to NULL. In contrast to *`PyArray_DiscardWritebackIfCopy`* it makes no attempt to copy the data from *obj->base* This undoes *`PyArray_SetWritebackIfCopyBase`*. Usually this is called after an error when you are finished with `obj`, just before `Py_DECREF(obj)`. It may be called multiple times, or with `NULL` input.

**PyArray_XDECREF_ERR**(PyObject* *obj*)
Deprecated in 1.14, use *`PyArray_DiscardWritebackIfCopy`* followed by `Py_XDECREF`

DECREF's an array object which may have the (deprecated) *`NPY_ARRAY_UPDATEIFCOPY`* or *`NPY_ARRAY_WRITEBACKIFCOPY`* flag set without causing the contents to be copied back into the original array. Resets the *`NPY_ARRAY_WRITEABLE`* flag on the base object. This is useful for recovering from an error condition when writeback semantics are used, but will lead to wrong results.

### Enumerated Types

**NPY_SORTKIND**
A special variable-type which can take on different values to indicate the sorting algorithm being used.

**NPY_QUICKSORT**

**NPY_HEAPSORT**

**NPY_MERGESORT**

**NPY_STABLESORT**
Used as an alias of *`NPY_MERGESORT`* and vica versa.

**NPY_NSORTS**
Defined to be the number of sorts. It is fixed at three by the need for backwards compatibility, and consequently *NPY_MERGESORT* and *NPY_STABLESORT* are aliased to each other and may refer to one of several stable sorting algorithms depending on the data type.

**NPY_SCALARKIND**
A special variable type indicating the number of "kinds" of scalars distinguished in determining scalar-coercion rules. This variable can take on the values:

**NPY_NOSCALAR**

**NPY_BOOL_SCALAR**

**NPY_INTPOS_SCALAR**

**NPY_INTNEG_SCALAR**

**NPY_FLOAT_SCALAR**

**NPY_COMPLEX_SCALAR**

**NPY_OBJECT_SCALAR**

**NPY_NSCALARKINDS**
Defined to be the number of scalar kinds (not including *NPY_NOSCALAR*).

**NPY_ORDER**
An enumeration type indicating the element order that an array should be interpreted in. When a brand new array is created, generally only **NPY_CORDER** and **NPY_FORTRANORDER** are used, whereas when one or more inputs are provided, the order can be based on them.

**NPY_ANYORDER**
Fortran order if all the inputs are Fortran, C otherwise.

**NPY_CORDER**
C order.

**NPY_FORTRANORDER**
Fortran order.

**NPY_KEEPORDER**
An order as close to the order of the inputs as possible, even if the input is in neither C nor Fortran order.

**NPY_CLIPMODE**
A variable type indicating the kind of clipping that should be applied in certain functions.

**NPY_RAISE**
The default for most operations, raises an exception if an index is out of bounds.

**NPY_CLIP**
Clips an index to the valid range if it is out of bounds.

**NPY_WRAP**
Wraps an index to the valid range if it is out of bounds.

**NPY_CASTING**
New in version 1.6.

An enumeration type indicating how permissive data conversions should be. This is used by the iterator added in NumPy 1.6, and is intended to be used more broadly in a future version.

**NPY_NO_CASTING**
Only allow identical types.

**NPY_EQUIV_CASTING**
Allow identical and casts involving byte swapping.

**NPY_SAFE_CASTING**
Only allow casts which will not cause values to be rounded, truncated, or otherwise changed.

**NPY_SAME_KIND_CASTING**
Allow any safe casts, and casts between types of the same kind. For example, float64 -> float32 is permitted with this rule.

**NPY_UNSAFE_CASTING**
Allow any cast, no matter what kind of data loss may occur.

## 8.5 Array Iterator API

New in version 1.6.

### 8.5.1 Array Iterator

The array iterator encapsulates many of the key features in ufuncs, allowing user code to support features like output parameters, preservation of memory layouts, and buffering of data with the wrong alignment or type, without requiring difficult coding.

This page documents the API for the iterator. The iterator is named `NpyIter` and functions are named `NpyIter_*`.

There is an *introductory guide to array iteration* which may be of interest for those using this C API. In many instances, testing out ideas by creating the iterator in Python is a good idea before writing the C iteration code.

### 8.5.2 Simple Iteration Example

The best way to become familiar with the iterator is to look at its usage within the NumPy codebase itself. For example, here is a slightly tweaked version of the code for `PyArray_CountNonzero`, which counts the number of non-zero elements in an array.

```
npy_intp PyArray_CountNonzero(PyArrayObject* self)
{
    /* Nonzero boolean function */
    PyArray_NonzeroFunc* nonzero = PyArray_DESCR(self)->f->nonzero;

    NpyIter* iter;
    NpyIter_IterNextFunc *iternext;
    char** dataptr;
    npy_intp nonzero_count;
    npy_intp* strideptr,* innersizeptr;

    /* Handle zero-sized arrays specially */
    if (PyArray_SIZE(self) == 0) {
        return 0;
    }

    /*
     * Create and use an iterator to count the nonzeros.
     *   flag NPY_ITER_READONLY
     *     - The array is never written to.
```

(continues on next page)

(continued from previous page)

```
     *   flag NPY_ITER_EXTERNAL_LOOP
     *     - Inner loop is done outside the iterator for efficiency.
     *   flag NPY_ITER_NPY_ITER_REFS_OK
     *     - Reference types are acceptable.
     *   order NPY_KEEPORDER
     *     - Visit elements in memory order, regardless of strides.
     *       This is good for performance when the specific order
     *       elements are visited is unimportant.
     *   casting NPY_NO_CASTING
     *     - No casting is required for this operation.
     */
    iter = NpyIter_New(self, NPY_ITER_READONLY|
                             NPY_ITER_EXTERNAL_LOOP|
                             NPY_ITER_REFS_OK,
                        NPY_KEEPORDER, NPY_NO_CASTING,
                        NULL);
    if (iter == NULL) {
        return -1;
    }

    /*
     * The iternext function gets stored in a local variable
     * so it can be called repeatedly in an efficient manner.
     */
    iternext = NpyIter_GetIterNext(iter, NULL);
    if (iternext == NULL) {
        NpyIter_Deallocate(iter);
        return -1;
    }
    /* The location of the data pointer which the iterator may update */
    dataptr = NpyIter_GetDataPtrArray(iter);
    /* The location of the stride which the iterator may update */
    strideptr = NpyIter_GetInnerStrideArray(iter);
    /* The location of the inner loop size which the iterator may update */
    innersizeptr = NpyIter_GetInnerLoopSizePtr(iter);

    nonzero_count = 0;
    do {
        /* Get the inner loop data/stride/count values */
        char* data = *dataptr;
        npy_intp stride = *strideptr;
        npy_intp count = *innersizeptr;

        /* This is a typical inner loop for NPY_ITER_EXTERNAL_LOOP */
        while (count--) {
            if (nonzero(data, self)) {
                ++nonzero_count;
            }
            data += stride;
        }

        /* Increment the iterator to the next inner loop */
    } while(iternext(iter));

    NpyIter_Deallocate(iter);

    return nonzero_count;
```

(continues on next page)

(continued from previous page)

```
}
```

## 8.5.3 Simple Multi-Iteration Example

Here is a simple copy function using the iterator. The `order` parameter is used to control the memory layout of the allocated result, typically *`NPY_KEEPORDER`* is desired.

```
PyObject *CopyArray(PyObject *arr, NPY_ORDER order)
{
    NpyIter *iter;
    NpyIter_IterNextFunc *iternext;
    PyObject *op[2], *ret;
    npy_uint32 flags;
    npy_uint32 op_flags[2];
    npy_intp itemsize, *innersizeptr, innerstride;
    char **dataptrarray;

    /*
     * No inner iteration - inner loop is handled by CopyArray code
     */
    flags = NPY_ITER_EXTERNAL_LOOP;
    /*
     * Tell the constructor to automatically allocate the output.
     * The data type of the output will match that of the input.
     */
    op[0] = arr;
    op[1] = NULL;
    op_flags[0] = NPY_ITER_READONLY;
    op_flags[1] = NPY_ITER_WRITEONLY | NPY_ITER_ALLOCATE;

    /* Construct the iterator */
    iter = NpyIter_MultiNew(2, op, flags, order, NPY_NO_CASTING,
                            op_flags, NULL);
    if (iter == NULL) {
        return NULL;
    }

    /*
     * Make a copy of the iternext function pointer and
     * a few other variables the inner loop needs.
     */
    iternext = NpyIter_GetIterNext(iter, NULL);
    innerstride = NpyIter_GetInnerStrideArray(iter)[0];
    itemsize = NpyIter_GetDescrArray(iter)[0]->elsize;
    /*
     * The inner loop size and data pointers may change during the
     * loop, so just cache the addresses.
     */
    innersizeptr = NpyIter_GetInnerLoopSizePtr(iter);
    dataptrarray = NpyIter_GetDataPtrArray(iter);

    /*
     * Note that because the iterator allocated the output,
     * it matches the iteration order and is packed tightly,
     * so we don't need to check it like the input.
```

(continues on next page)

(continued from previous page)

```
     */
    if (innerstride == itemsize) {
        do {
            memcpy(dataptrarray[1], dataptrarray[0],
                                        itemsize * (*innersizeptr));
        } while (iternext(iter));
    } else {
        /* For efficiency, should specialize this based on item size... */
        npy_intp i;
        do {
            npy_intp size = *innersizeptr;
            char *src = dataptrarray[0], *dst = dataptrarray[1];
            for(i = 0; i < size; i++, src += innerstride, dst += itemsize) {
                memcpy(dst, src, itemsize);
            }
        } while (iternext(iter));
    }

    /* Get the result from the iterator object array */
    ret = NpyIter_GetOperandArray(iter)[1];
    Py_INCREF(ret);

    if (NpyIter_Deallocate(iter) != NPY_SUCCEED) {
        Py_DECREF(ret);
        return NULL;
    }

    return ret;
}
```

## 8.5.4 Iterator Data Types

The iterator layout is an internal detail, and user code only sees an incomplete struct.

**`NpyIter`**
This is an opaque pointer type for the iterator. Access to its contents can only be done through the iterator API.

**`NpyIter_Type`**
This is the type which exposes the iterator to Python. Currently, no API is exposed which provides access to the values of a Python-created iterator. If an iterator is created in Python, it must be used in Python and vice versa. Such an API will likely be created in a future version.

**`NpyIter_IterNextFunc`**
This is a function pointer for the iteration loop, returned by *`NpyIter_GetIterNext`*.

**`NpyIter_GetMultiIndexFunc`**
This is a function pointer for getting the current iterator multi-index, returned by *`NpyIter_GetGetMultiIndex`*.

### 8.5.5 Construction and Destruction

*NpyIter** **`NpyIter_New`**(*PyArrayObject** *op*, *npy_uint32 flags*, *NPY_ORDER order*, *NPY_CASTING casting*, *PyArray_Descr** *dtype*)

Creates an iterator for the given numpy array object `op`.

Flags that may be passed in `flags` are any combination of the global and per-operand flags documented in *`NpyIter_MultiNew`*, except for *`NPY_ITER_ALLOCATE`*.

Any of the *`NPY_ORDER`* enum values may be passed to `order`. For efficient iteration, *`NPY_KEEPORDER`* is the best option, and the other orders enforce the particular iteration pattern.

Any of the *`NPY_CASTING`* enum values may be passed to `casting`. The values include *`NPY_NO_CASTING`*, *`NPY_EQUIV_CASTING`*, *`NPY_SAFE_CASTING`*, *`NPY_SAME_KIND_CASTING`*, and *`NPY_UNSAFE_CASTING`*. To allow the casts to occur, copying or buffering must also be enabled.

If `dtype` isn't `NULL`, then it requires that data type. If copying is allowed, it will make a temporary copy if the data is castable. If *`NPY_ITER_UPDATEIFCOPY`* is enabled, it will also copy the data back with another cast upon iterator destruction.

Returns NULL if there is an error, otherwise returns the allocated iterator.

To make an iterator similar to the old iterator, this should work.

```
iter = NpyIter_New(op, NPY_ITER_READWRITE,
                    NPY_CORDER, NPY_NO_CASTING, NULL);
```

If you want to edit an array with aligned `double` code, but the order doesn't matter, you would use this.

```
dtype = PyArray_DescrFromType(NPY_DOUBLE);
iter = NpyIter_New(op, NPY_ITER_READWRITE|
                    NPY_ITER_BUFFERED|
                    NPY_ITER_NBO|
                    NPY_ITER_ALIGNED,
                    NPY_KEEPORDER,
                    NPY_SAME_KIND_CASTING,
                    dtype);
Py_DECREF(dtype);
```

*NpyIter** **`NpyIter_MultiNew`**(*npy_intp nop*, *PyArrayObject*** *op*, *npy_uint32 flags*, *NPY_ORDER order*, *NPY_CASTING casting*, *npy_uint32** *op_flags*, *PyArray_Descr*** *op_dtypes*)

Creates an iterator for broadcasting the `nop` array objects provided in `op`, using regular NumPy broadcasting rules.

Any of the *`NPY_ORDER`* enum values may be passed to `order`. For efficient iteration, *`NPY_KEEPORDER`* is the best option, and the other orders enforce the particular iteration pattern. When using *`NPY_KEEPORDER`*, if you also want to ensure that the iteration is not reversed along an axis, you should pass the flag *`NPY_ITER_DONT_NEGATE_STRIDES`*.

Any of the *`NPY_CASTING`* enum values may be passed to `casting`. The values include *`NPY_NO_CASTING`*, *`NPY_EQUIV_CASTING`*, *`NPY_SAFE_CASTING`*, *`NPY_SAME_KIND_CASTING`*, and *`NPY_UNSAFE_CASTING`*. To allow the casts to occur, copying or buffering must also be enabled.

If `op_dtypes` isn't `NULL`, it specifies a data type or `NULL` for each `op[i]`.

Returns NULL if there is an error, otherwise returns the allocated iterator.

Flags that may be passed in `flags`, applying to the whole iterator, are:

**`NPY_ITER_C_INDEX`**

Causes the iterator to track a raveled flat index matching C order. This option cannot be used with *`NPY_ITER_F_INDEX`*.

**NPY_ITER_F_INDEX**

Causes the iterator to track a raveled flat index matching Fortran order. This option cannot be used with *NPY_ITER_C_INDEX*.

**NPY_ITER_MULTI_INDEX**

Causes the iterator to track a multi-index. This prevents the iterator from coalescing axes to produce bigger inner loops. If the loop is also not buffered and no index is being tracked (*NpyIter_RemoveAxis* can be called), then the iterator size can be `-1` to indicate that the iterator is too large. This can happen due to complex broadcasting and will result in errors being created when the setting the iterator range, removing the multi index, or getting the next function. However, it is possible to remove axes again and use the iterator normally if the size is small enough after removal.

**NPY_ITER_EXTERNAL_LOOP**

Causes the iterator to skip iteration of the innermost loop, requiring the user of the iterator to handle it.

This flag is incompatible with *NPY_ITER_C_INDEX*, *NPY_ITER_F_INDEX*, and *NPY_ITER_MULTI_INDEX*.

**NPY_ITER_DONT_NEGATE_STRIDES**

This only affects the iterator when *NPY_KEEPORDER* is specified for the order parameter. By default with *NPY_KEEPORDER*, the iterator reverses axes which have negative strides, so that memory is traversed in a forward direction. This disables this step. Use this flag if you want to use the underlying memory-ordering of the axes, but don't want an axis reversed. This is the behavior of `numpy.ravel(a, order='K')`, for instance.

**NPY_ITER_COMMON_DTYPE**

Causes the iterator to convert all the operands to a common data type, calculated based on the ufunc type promotion rules. Copying or buffering must be enabled.

If the common data type is known ahead of time, don't use this flag. Instead, set the requested dtype for all the operands.

**NPY_ITER_REFS_OK**

Indicates that arrays with reference types (object arrays or structured arrays containing an object type) may be accepted and used in the iterator. If this flag is enabled, the caller must be sure to check whether `NpyIter_IterationNeedsAPI(iter)` is true, in which case it may not release the GIL during iteration.

**NPY_ITER_ZEROSIZE_OK**

Indicates that arrays with a size of zero should be permitted. Since the typical iteration loop does not naturally work with zero-sized arrays, you must check that the IterSize is larger than zero before entering the iteration loop. Currently only the operands are checked, not a forced shape.

**NPY_ITER_REDUCE_OK**

Permits writeable operands with a dimension with zero stride and size greater than one. Note that such operands must be read/write.

When buffering is enabled, this also switches to a special buffering mode which reduces the loop length as necessary to not trample on values being reduced.

Note that if you want to do a reduction on an automatically allocated output, you must use *NpyIter_GetOperandArray* to get its reference, then set every value to the reduction unit before doing the iteration loop. In the case of a buffered reduction, this means you must also specify the flag *NPY_ITER_DELAY_BUFALLOC*, then reset the iterator after initializing the allocated operand to prepare the buffers.

**NPY_ITER_RANGED**

Enables support for iteration of sub-ranges of the full `iterindex` range `[0, NpyIter_IterSize(iter))`. Use the function *`NpyIter_ResetToIterIndexRange`* to specify a range for iteration.

This flag can only be used with *`NPY_ITER_EXTERNAL_LOOP`* when *`NPY_ITER_BUFFERED`* is enabled. This is because without buffering, the inner loop is always the size of the innermost iteration dimension, and allowing it to get cut up would require special handling, effectively making it more like the buffered version.

**`NPY_ITER_BUFFERED`**

Causes the iterator to store buffering data, and use buffering to satisfy data type, alignment, and byte-order requirements. To buffer an operand, do not specify the *`NPY_ITER_COPY`* or *`NPY_ITER_UPDATEIFCOPY`* flags, because they will override buffering. Buffering is especially useful for Python code using the iterator, allowing for larger chunks of data at once to amortize the Python interpreter overhead.

If used with *`NPY_ITER_EXTERNAL_LOOP`*, the inner loop for the caller may get larger chunks than would be possible without buffering, because of how the strides are laid out.

Note that if an operand is given the flag *`NPY_ITER_COPY`* or *`NPY_ITER_UPDATEIFCOPY`*, a copy will be made in preference to buffering. Buffering will still occur when the array was broadcast so elements need to be duplicated to get a constant stride.

In normal buffering, the size of each inner loop is equal to the buffer size, or possibly larger if *`NPY_ITER_GROWINNER`* is specified. If *`NPY_ITER_REDUCE_OK`* is enabled and a reduction occurs, the inner loops may become smaller depending on the structure of the reduction.

**`NPY_ITER_GROWINNER`**

When buffering is enabled, this allows the size of the inner loop to grow when buffering isn't necessary. This option is best used if you're doing a straight pass through all the data, rather than anything with small cache-friendly arrays of temporary values for each inner loop.

**`NPY_ITER_DELAY_BUFALLOC`**

When buffering is enabled, this delays allocation of the buffers until *`NpyIter_Reset`* or another reset function is called. This flag exists to avoid wasteful copying of buffer data when making multiple copies of a buffered iterator for multi-threaded iteration.

Another use of this flag is for setting up reduction operations. After the iterator is created, and a reduction output is allocated automatically by the iterator (be sure to use READWRITE access), its value may be initialized to the reduction unit. Use *`NpyIter_GetOperandArray`* to get the object. Then, call *`NpyIter_Reset`* to allocate and fill the buffers with their initial values.

**`NPY_ITER_COPY_IF_OVERLAP`**

If any write operand has overlap with any read operand, eliminate all overlap by making temporary copies (enabling UPDATEIFCOPY for write operands, if necessary). A pair of operands has overlap if there is a memory address that contains data common to both arrays.

Because exact overlap detection has exponential runtime in the number of dimensions, the decision is made based on heuristics, which has false positives (needless copies in unusual cases) but has no false negatives.

If any read/write overlap exists, this flag ensures the result of the operation is the same as if all operands were copied. In cases where copies would need to be made, **the result of the computation may be undefined without this flag!**

Flags that may be passed in `op_flags[i]`, where `0 <= i < nop`:

**`NPY_ITER_READWRITE`**

**`NPY_ITER_READONLY`**

**NPY_ITER_WRITEONLY**

Indicate how the user of the iterator will read or write to `op[i]`. Exactly one of these flags must be specified per operand. Using `NPY_ITER_READWRITE` or `NPY_ITER_WRITEONLY` for a user-provided operand may trigger *WRITEBACKIFCOPY`* semantics. The data will be written back to the original array when `NpyIter_Deallocate` is called.

**NPY_ITER_COPY**

Allow a copy of `op[i]` to be made if it does not meet the data type or alignment requirements as specified by the constructor flags and parameters.

**NPY_ITER_UPDATEIFCOPY**

Triggers *`NPY_ITER_COPY`*, and when an array operand is flagged for writing and is copied, causes the data in a copy to be copied back to `op[i]` when `NpyIter_Deallocate` is called.

If the operand is flagged as write-only and a copy is needed, an uninitialized temporary array will be created and then copied to back to `op[i]` on calling `NpyIter_Deallocate`, instead of doing the unnecessary copy operation.

**NPY_ITER_NBO**

**NPY_ITER_ALIGNED**

**NPY_ITER_CONTIG**

Causes the iterator to provide data for `op[i]` that is in native byte order, aligned according to the dtype requirements, contiguous, or any combination.

By default, the iterator produces pointers into the arrays provided, which may be aligned or unaligned, and with any byte order. If copying or buffering is not enabled and the operand data doesn't satisfy the constraints, an error will be raised.

The contiguous constraint applies only to the inner loop, successive inner loops may have arbitrary pointer changes.

If the requested data type is in non-native byte order, the NBO flag overrides it and the requested data type is converted to be in native byte order.

**NPY_ITER_ALLOCATE**

This is for output arrays, and requires that the flag *`NPY_ITER_WRITEONLY`* or *`NPY_ITER_READWRITE`* be set. If `op[i]` is NULL, creates a new array with the final broadcast dimensions, and a layout matching the iteration order of the iterator.

When `op[i]` is NULL, the requested data type `op_dtypes[i]` may be NULL as well, in which case it is automatically generated from the dtypes of the arrays which are flagged as readable. The rules for generating the dtype are the same is for UFuncs. Of special note is handling of byte order in the selected dtype. If there is exactly one input, the input's dtype is used as is. Otherwise, if more than one input dtypes are combined together, the output will be in native byte order.

After being allocated with this flag, the caller may retrieve the new array by calling *`NpyIter_GetOperandArray`* and getting the i-th object in the returned C array. The caller must call Py_INCREF on it to claim a reference to the array.

**NPY_ITER_NO_SUBTYPE**

For use with *`NPY_ITER_ALLOCATE`*, this flag disables allocating an array subtype for the output, forcing it to be a straight ndarray.

TODO: Maybe it would be better to introduce a function `NpyIter_GetWrappedOutput` and remove this flag?

**NPY_ITER_NO_BROADCAST**

Ensures that the input or output matches the iteration dimensions exactly.

**`NPY_ITER_ARRAYMASK`**

New in version 1.7.

Indicates that this operand is the mask to use for selecting elements when writing to operands which have the *`NPY_ITER_WRITEMASKED`* flag applied to them. Only one operand may have *`NPY_ITER_ARRAYMASK`* flag applied to it.

The data type of an operand with this flag should be either *`NPY_BOOL`*, *`NPY_MASK`*, or a struct dtype whose fields are all valid mask dtypes. In the latter case, it must match up with a struct operand being WRITEMASKED, as it is specifying a mask for each field of that array.

This flag only affects writing from the buffer back to the array. This means that if the operand is also *`NPY_ITER_READWRITE`* or *`NPY_ITER_WRITEONLY`*, code doing iteration can write to this operand to control which elements will be untouched and which ones will be modified. This is useful when the mask should be a combination of input masks.

**`NPY_ITER_WRITEMASKED`**

New in version 1.7.

This array is the mask for all *`writemasked`* operands. Code uses the `writemasked` flag which indicates that only elements where the chosen ARRAYMASK operand is True will be written to. In general, the iterator does not enforce this, it is up to the code doing the iteration to follow that promise.

When `writemasked` flag is used, and this operand is buffered, this changes how data is copied from the buffer into the array. A masked copying routine is used, which only copies the elements in the buffer for which `writemasked` returns true from the corresponding element in the ARRAYMASK operand.

**`NPY_ITER_OVERLAP_ASSUME_ELEMENTWISE`**

In memory overlap checks, assume that operands with `NPY_ITER_OVERLAP_ASSUME_ELEMENTWISE` enabled are accessed only in the iterator order.

This enables the iterator to reason about data dependency, possibly avoiding unnecessary copies.

This flag has effect only if `NPY_ITER_COPY_IF_OVERLAP` is enabled on the iterator.

*NpyIter** **`NpyIter_AdvancedNew`** (*npy_intp nop*, *PyArrayObject** op*, *npy_uint32 flags*, *NPY_ORDER order*, *NPY_CASTING casting*, *npy_uint32* op_flags*, *PyArray_Descr** op_dtypes*, int *oa_ndim*, int** *op_axes*, *npy_intp* const* *itershape*, *npy_intp buffersize*)

Extends *`NpyIter_MultiNew`* with several advanced options providing more control over broadcasting and buffering.

If -1/NULL values are passed to `oa_ndim`, `op_axes`, `itershape`, and `buffersize`, it is equivalent to *`NpyIter_MultiNew`*.

The parameter `oa_ndim`, when not zero or -1, specifies the number of dimensions that will be iterated with customized broadcasting. If it is provided, `op_axes` must and `itershape` can also be provided. The `op_axes` parameter let you control in detail how the axes of the operand arrays get matched together and iterated. In `op_axes`, you must provide an array of `nop` pointers to `oa_ndim`-sized arrays of type `npy_intp`. If an entry in `op_axes` is NULL, normal broadcasting rules will apply. In `op_axes[j][i]` is stored either a valid axis of `op[j]`, or -1 which means `newaxis`. Within each `op_axes[j]` array, axes may not be repeated. The following example is how normal broadcasting applies to a 3-D array, a 2-D array, a 1-D array and a scalar.

**Note**: Before NumPy 1.8 `` oa_ndim == 0` was used for signalling that that ``op_axes`` and `itershape` are unused. This is deprecated and should be replaced with -1. Better backward compatibility may be achieved by using *`NpyIter_MultiNew`* for this case.

```
int oa_ndim = 3;               /* # iteration axes */
int op0_axes[] = {0, 1, 2};    /* 3-D operand */
int op1_axes[] = {-1, 0, 1};   /* 2-D operand */
int op2_axes[] = {-1, -1, 0};  /* 1-D operand */
int op3_axes[] = {-1, -1, -1}  /* 0-D (scalar) operand */
int* op_axes[] = {op0_axes, op1_axes, op2_axes, op3_axes};
```

The `itershape` parameter allows you to force the iterator to have a specific iteration shape. It is an array of length `oa_ndim`. When an entry is negative, its value is determined from the operands. This parameter allows automatically allocated outputs to get additional dimensions which don't match up with any dimension of an input.

If `buffersize` is zero, a default buffer size is used, otherwise it specifies how big of a buffer to use. Buffers which are powers of 2 such as 4096 or 8192 are recommended.

Returns NULL if there is an error, otherwise returns the allocated iterator.

*NpyIter** **`NpyIter_Copy`** (*NpyIter* iter*)

Makes a copy of the given iterator. This function is provided primarily to enable multi-threaded iteration of the data.

*TODO*: Move this to a section about multithreaded iteration.

The recommended approach to multithreaded iteration is to first create an iterator with the flags `NPY_ITER_EXTERNAL_LOOP`, `NPY_ITER_RANGED`, `NPY_ITER_BUFFERED`, `NPY_ITER_DELAY_BUFALLOC`, and possibly `NPY_ITER_GROWINNER`. Create a copy of this iterator for each thread (minus one for the first iterator). Then, take the iteration index range `[0, NpyIter_GetIterSize(iter))` and split it up into tasks, for example using a TBB parallel_for loop. When a thread gets a task to execute, it then uses its copy of the iterator by calling `NpyIter_ResetToIterIndexRange` and iterating over the full range.

When using the iterator in multi-threaded code or in code not holding the Python GIL, care must be taken to only call functions which are safe in that context. `NpyIter_Copy` cannot be safely called without the Python GIL, because it increments Python references. The `Reset*` and some other functions may be safely called by passing in the `errmsg` parameter as non-NULL, so that the functions will pass back errors through it instead of setting a Python exception.

`NpyIter_Deallocate` must be called for each copy.

**`int NpyIter_RemoveAxis(NpyIter* iter, int axis)``**

Removes an axis from iteration. This requires that `NPY_ITER_MULTI_INDEX` was set for iterator creation, and does not work if buffering is enabled or an index is being tracked. This function also resets the iterator to its initial state.

This is useful for setting up an accumulation loop, for example. The iterator can first be created with all the dimensions, including the accumulation axis, so that the output gets created correctly. Then, the accumulation axis can be removed, and the calculation done in a nested fashion.

**WARNING**: This function may change the internal memory layout of the iterator. Any cached functions or pointers from the iterator must be retrieved again! The iterator range will be reset as well.

Returns `NPY_SUCCEED` or `NPY_FAIL`.

int **`NpyIter_RemoveMultiIndex`** (*NpyIter* iter*)

If the iterator is tracking a multi-index, this strips support for them, and does further iterator optimizations that are possible if multi-indices are not needed. This function also resets the iterator to its initial state.

**WARNING**: This function may change the internal memory layout of the iterator. Any cached functions or pointers from the iterator must be retrieved again!

After calling this function, `NpyIter_HasMultiIndex(iter)` will return false.

Returns `NPY_SUCCEED` or `NPY_FAIL`.

int **`NpyIter_EnableExternalLoop`** (*NpyIter** *iter*)

If *`NpyIter_RemoveMultiIndex`* was called, you may want to enable the flag *`NPY_ITER_EXTERNAL_LOOP`*. This flag is not permitted together with *`NPY_ITER_MULTI_INDEX`*, so this function is provided to enable the feature after *`NpyIter_RemoveMultiIndex`* is called. This function also resets the iterator to its initial state.

**WARNING**: This function changes the internal logic of the iterator. Any cached functions or pointers from the iterator must be retrieved again!

Returns `NPY_SUCCEED` or `NPY_FAIL`.

int **`NpyIter_Deallocate`** (*NpyIter** *iter*)

Deallocates the iterator object and resolves any needed writebacks.

Returns `NPY_SUCCEED` or `NPY_FAIL`.

int **`NpyIter_Reset`** (*NpyIter** *iter*, char** *errmsg*)

Resets the iterator back to its initial state, at the beginning of the iteration range.

Returns `NPY_SUCCEED` or `NPY_FAIL`. If errmsg is non-NULL, no Python exception is set when `NPY_FAIL` is returned. Instead, *errmsg is set to an error message. When errmsg is non-NULL, the function may be safely called without holding the Python GIL.

int **`NpyIter_ResetToIterIndexRange`** (*NpyIter** *iter*, *npy_intp istart*, *npy_intp iend*, char** *errmsg*)

Resets the iterator and restricts it to the `iterindex` range `[istart, iend)`. See *`NpyIter_Copy`* for an explanation of how to use this for multi-threaded iteration. This requires that the flag *`NPY_ITER_RANGED`* was passed to the iterator constructor.

If you want to reset both the `iterindex` range and the base pointers at the same time, you can do the following to avoid extra buffer copying (be sure to add the return code error checks when you copy this code).

```
/* Set to a trivial empty range */
NpyIter_ResetToIterIndexRange(iter, 0, 0);
/* Set the base pointers */
NpyIter_ResetBasePointers(iter, baseptrs);
/* Set to the desired range */
NpyIter_ResetToIterIndexRange(iter, istart, iend);
```

Returns `NPY_SUCCEED` or `NPY_FAIL`. If errmsg is non-NULL, no Python exception is set when `NPY_FAIL` is returned. Instead, *errmsg is set to an error message. When errmsg is non-NULL, the function may be safely called without holding the Python GIL.

int **`NpyIter_ResetBasePointers`** (*NpyIter *iter*, char** *baseptrs*, char** *errmsg*)

Resets the iterator back to its initial state, but using the values in `baseptrs` for the data instead of the pointers from the arrays being iterated. This functions is intended to be used, together with the `op_axes` parameter, by nested iteration code with two or more iterators.

Returns `NPY_SUCCEED` or `NPY_FAIL`. If errmsg is non-NULL, no Python exception is set when `NPY_FAIL` is returned. Instead, *errmsg is set to an error message. When errmsg is non-NULL, the function may be safely called without holding the Python GIL.

*TODO*: Move the following into a special section on nested iterators.

Creating iterators for nested iteration requires some care. All the iterator operands must match exactly, or the calls to *`NpyIter_ResetBasePointers`* will be invalid. This means that automatic copies and output allocation should not be used haphazardly. It is possible to still use the automatic data conversion and casting features of the iterator by creating one of the iterators with all the conversion parameters enabled, then grabbing the allocated operands with the *`NpyIter_GetOperandArray`* function and passing them into the constructors for the rest of the iterators.

**WARNING**: When creating iterators for nested iteration, the code must not use a dimension more than once in the different iterators. If this is done, nested iteration will produce out-of-bounds pointers during iteration.

**WARNING**: When creating iterators for nested iteration, buffering can only be applied to the innermost iterator. If a buffered iterator is used as the source for `baseptrs`, it will point into a small buffer instead of the array and the inner iteration will be invalid.

The pattern for using nested iterators is as follows.

```
NpyIter *iter1, *iter1;
NpyIter_IterNextFunc *iternext1, *iternext2;
char **dataptrs1;

/*
 * With the exact same operands, no copies allowed, and
 * no axis in op_axes used both in iter1 and iter2.
 * Buffering may be enabled for iter2, but not for iter1.
 */
iter1 = ...; iter2 = ...;

iternext1 = NpyIter_GetIterNext(iter1);
iternext2 = NpyIter_GetIterNext(iter2);
dataptrs1 = NpyIter_GetDataPtrArray(iter1);

do {
    NpyIter_ResetBasePointers(iter2, dataptrs1);
    do {
        /* Use the iter2 values */
    } while (iternext2(iter2));
} while (iternext1(iter1));
```

int **`NpyIter_GotoMultiIndex`** (*NpyIter** *iter*, *npy_intp* const* *multi_index*)

Adjusts the iterator to point to the `ndim` indices pointed to by `multi_index`. Returns an error if a multi-index is not being tracked, the indices are out of bounds, or inner loop iteration is disabled.

Returns `NPY_SUCCEED` or `NPY_FAIL`.

int **`NpyIter_GotoIndex`** (*NpyIter** *iter*, *npy_intp* *index*)

Adjusts the iterator to point to the `index` specified. If the iterator was constructed with the flag *`NPY_ITER_C_INDEX`*, `index` is the C-order index, and if the iterator was constructed with the flag *`NPY_ITER_F_INDEX`*, `index` is the Fortran-order index. Returns an error if there is no index being tracked, the index is out of bounds, or inner loop iteration is disabled.

Returns `NPY_SUCCEED` or `NPY_FAIL`.

*npy_intp* **`NpyIter_GetIterSize`** (*NpyIter** *iter*)

Returns the number of elements being iterated. This is the product of all the dimensions in the shape. When a multi index is being tracked (and *NpyIter_RemoveAxis* may be called) the size may be `-1` to indicate an iterator is too large. Such an iterator is invalid, but may become valid after *NpyIter_RemoveAxis* is called. It is not necessary to check for this case.

*npy_intp* **`NpyIter_GetIterIndex`** (*NpyIter** *iter*)

Gets the `iterindex` of the iterator, which is an index matching the iteration order of the iterator.

void **`NpyIter_GetIterIndexRange`** (*NpyIter** *iter*, *npy_intp** *istart*, *npy_intp** *iend*)

Gets the `iterindex` sub-range that is being iterated. If *`NPY_ITER_RANGED`* was not specified, this always returns the range `[0, NpyIter_IterSize(iter))`.

int **`NpyIter_GotoIterIndex`** (*NpyIter** *iter*, *npy_intp* *iterindex*)

Adjusts the iterator to point to the `iterindex` specified. The IterIndex is an index matching the iteration order of the iterator. Returns an error if the `iterindex` is out of bounds, buffering is enabled, or inner loop iteration is disabled.

Returns `NPY_SUCCEED` or `NPY_FAIL`.

*npy_bool* **`NpyIter_HasDelayedBufAlloc`** (*NpyIter* iter*)

Returns 1 if the flag *`NPY_ITER_DELAY_BUFALLOC`* was passed to the iterator constructor, and no call to one of the Reset functions has been done yet, 0 otherwise.

*npy_bool* **`NpyIter_HasExternalLoop`** (*NpyIter* iter*)

Returns 1 if the caller needs to handle the inner-most 1-dimensional loop, or 0 if the iterator handles all looping. This is controlled by the constructor flag *`NPY_ITER_EXTERNAL_LOOP`* or *`NpyIter_EnableExternalLoop`*.

*npy_bool* **`NpyIter_HasMultiIndex`** (*NpyIter* iter*)

Returns 1 if the iterator was created with the *`NPY_ITER_MULTI_INDEX`* flag, 0 otherwise.

*npy_bool* **`NpyIter_HasIndex`** (*NpyIter* iter*)

Returns 1 if the iterator was created with the *`NPY_ITER_C_INDEX`* or *`NPY_ITER_F_INDEX`* flag, 0 otherwise.

*npy_bool* **`NpyIter_RequiresBuffering`** (*NpyIter* iter*)

Returns 1 if the iterator requires buffering, which occurs when an operand needs conversion or alignment and so cannot be used directly.

*npy_bool* **`NpyIter_IsBuffered`** (*NpyIter* iter*)

Returns 1 if the iterator was created with the *`NPY_ITER_BUFFERED`* flag, 0 otherwise.

*npy_bool* **`NpyIter_IsGrowInner`** (*NpyIter* iter*)

Returns 1 if the iterator was created with the *`NPY_ITER_GROWINNER`* flag, 0 otherwise.

*npy_intp* **`NpyIter_GetBufferSize`** (*NpyIter* iter*)

If the iterator is buffered, returns the size of the buffer being used, otherwise returns 0.

int **`NpyIter_GetNDim`** (*NpyIter* iter*)

Returns the number of dimensions being iterated. If a multi-index was not requested in the iterator constructor, this value may be smaller than the number of dimensions in the original objects.

int **`NpyIter_GetNOp`** (*NpyIter* iter*)

Returns the number of operands in the iterator.

*npy_intp** **`NpyIter_GetAxisStrideArray`** (*NpyIter* iter*, int *axis*)

Gets the array of strides for the specified axis. Requires that the iterator be tracking a multi-index, and that buffering not be enabled.

This may be used when you want to match up operand axes in some fashion, then remove them with `NpyIter_RemoveAxis` to handle their processing manually. By calling this function before removing the axes, you can get the strides for the manual processing.

Returns `NULL` on error.

int **`NpyIter_GetShape`** (*NpyIter* iter*, *npy_intp* outshape*)

Returns the broadcast shape of the iterator in `outshape`. This can only be called on an iterator which is tracking a multi-index.

Returns `NPY_SUCCEED` or `NPY_FAIL`.

*PyArray_Descr*** **`NpyIter_GetDescrArray`** (*NpyIter* iter*)

This gives back a pointer to the `nop` data type Descrs for the objects being iterated. The result points into `iter`, so the caller does not gain any references to the Descrs.

This pointer may be cached before the iteration loop, calling `iternext` will not change it.

PyObject** **`NpyIter_GetOperandArray`** (*NpyIter* iter*)

This gives back a pointer to the `nop` operand PyObjects that are being iterated. The result points into `iter`, so the caller does not gain any references to the PyObjects.

PyObject* **`NpyIter_GetIterView`** (*NpyIter* iter*, *npy_intp i*)

This gives back a reference to a new ndarray view, which is a view into the i-th object in the array *`NpyIter_GetOperandArray`*, whose dimensions and strides match the internal optimized iteration pattern. A C-order iteration of this view is equivalent to the iterator's iteration order.

For example, if an iterator was created with a single array as its input, and it was possible to rearrange all its axes and then collapse it into a single strided iteration, this would return a view that is a one-dimensional array.

void **`NpyIter_GetReadFlags`** (*NpyIter* iter*, char* *outreadflags*)

Fills `nop` flags. Sets `outreadflags[i]` to 1 if `op[i]` can be read from, and to 0 if not.

void **`NpyIter_GetWriteFlags`** (*NpyIter* iter*, char* *outwriteflags*)

Fills `nop` flags. Sets `outwriteflags[i]` to 1 if `op[i]` can be written to, and to 0 if not.

int **`NpyIter_CreateCompatibleStrides`** (*NpyIter* iter*, *npy_intp itemsize*, *npy_intp* outstrides*)

Builds a set of strides which are the same as the strides of an output array created using the *`NPY_ITER_ALLOCATE`* flag, where NULL was passed for op_axes. This is for data packed contiguously, but not necessarily in C or Fortran order. This should be used together with *`NpyIter_GetShape`* and *`NpyIter_GetNDim`* with the flag *`NPY_ITER_MULTI_INDEX`* passed into the constructor.

A use case for this function is to match the shape and layout of the iterator and tack on one or more dimensions. For example, in order to generate a vector per input value for a numerical gradient, you pass in ndim*itemsize for itemsize, then add another dimension to the end with size ndim and stride itemsize. To do the Hessian matrix, you do the same thing but add two dimensions, or take advantage of the symmetry and pack it into 1 dimension with a particular encoding.

This function may only be called if the iterator is tracking a multi-index and if *`NPY_ITER_DONT_NEGATE_STRIDES`* was used to prevent an axis from being iterated in reverse order.

If an array is created with this method, simply adding 'itemsize' for each iteration will traverse the new array matching the iterator.

Returns `NPY_SUCCEED` or `NPY_FAIL`.

*npy_bool* **`NpyIter_IsFirstVisit`** (*NpyIter* iter*, int *iop*)

New in version 1.7.

Checks to see whether this is the first time the elements of the specified reduction operand which the iterator points at are being seen for the first time. The function returns a reasonable answer for reduction operands and when buffering is disabled. The answer may be incorrect for buffered non-reduction operands.

This function is intended to be used in EXTERNAL_LOOP mode only, and will produce some wrong answers when that mode is not enabled.

If this function returns true, the caller should also check the inner loop stride of the operand, because if that stride is 0, then only the first element of the innermost external loop is being visited for the first time.

*WARNING*: For performance reasons, 'iop' is not bounds-checked, it is not confirmed that 'iop' is actually a reduction operand, and it is not confirmed that EXTERNAL_LOOP mode is enabled. These checks are the responsibility of the caller, and should be done outside of any inner loops.

### 8.5.6 Functions For Iteration

*NpyIter_IterNextFunc** **NpyIter_GetIterNext** (*NpyIter** *iter*, char** *errmsg*)

Returns a function pointer for iteration. A specialized version of the function pointer may be calculated by this function instead of being stored in the iterator structure. Thus, to get good performance, it is required that the function pointer be saved in a variable rather than retrieved for each loop iteration.

Returns NULL if there is an error. If errmsg is non-NULL, no Python exception is set when `NPY_FAIL` is returned. Instead, *errmsg is set to an error message. When errmsg is non-NULL, the function may be safely called without holding the Python GIL.

The typical looping construct is as follows.

```
NpyIter_IterNextFunc *iternext = NpyIter_GetIterNext(iter, NULL);
char** dataptr = NpyIter_GetDataPtrArray(iter);

do {
    /* use the addresses dataptr[0], ... dataptr[nop-1] */
} while(iternext(iter));
```

When *`NPY_ITER_EXTERNAL_LOOP`* is specified, the typical inner loop construct is as follows.

```
NpyIter_IterNextFunc *iternext = NpyIter_GetIterNext(iter, NULL);
char** dataptr = NpyIter_GetDataPtrArray(iter);
npy_intp* stride = NpyIter_GetInnerStrideArray(iter);
npy_intp* size_ptr = NpyIter_GetInnerLoopSizePtr(iter), size;
npy_intp iop, nop = NpyIter_GetNOp(iter);

do {
    size = *size_ptr;
    while (size--) {
        /* use the addresses dataptr[0], ... dataptr[nop-1] */
        for (iop = 0; iop < nop; ++iop) {
            dataptr[iop] += stride[iop];
        }
    }
} while (iternext());
```

Observe that we are using the dataptr array inside the iterator, not copying the values to a local temporary. This is possible because when `iternext()` is called, these pointers will be overwritten with fresh values, not incrementally updated.

If a compile-time fixed buffer is being used (both flags *`NPY_ITER_BUFFERED`* and *`NPY_ITER_EXTERNAL_LOOP`*), the inner size may be used as a signal as well. The size is guaranteed to become zero when `iternext()` returns false, enabling the following loop construct. Note that if you use this construct, you should not pass *`NPY_ITER_GROWINNER`* as a flag, because it will cause larger sizes under some circumstances.

```
/* The constructor should have buffersize passed as this value */
#define FIXED_BUFFER_SIZE 1024

NpyIter_IterNextFunc *iternext = NpyIter_GetIterNext(iter, NULL);
char **dataptr = NpyIter_GetDataPtrArray(iter);
npy_intp *stride = NpyIter_GetInnerStrideArray(iter);
npy_intp *size_ptr = NpyIter_GetInnerLoopSizePtr(iter), size;
npy_intp i, iop, nop = NpyIter_GetNOp(iter);

/* One loop with a fixed inner size */
```

(continues on next page)

(continued from previous page)

```
size = *size_ptr;
while (size == FIXED_BUFFER_SIZE) {
    /*
     * This loop could be manually unrolled by a factor
     * which divides into FIXED_BUFFER_SIZE
     */
    for (i = 0; i < FIXED_BUFFER_SIZE; ++i) {
        /* use the addresses dataptr[0], ... dataptr[nop-1] */
        for (iop = 0; iop < nop; ++iop) {
            dataptr[iop] += stride[iop];
        }
    }
    iternext();
    size = *size_ptr;
}

/* Finish-up loop with variable inner size */
if (size > 0) do {
    size = *size_ptr;
    while (size--) {
        /* use the addresses dataptr[0], ... dataptr[nop-1] */
        for (iop = 0; iop < nop; ++iop) {
            dataptr[iop] += stride[iop];
        }
    }
} while (iternext());
```

*NpyIter_GetMultiIndexFunc* ***NpyIter_GetGetMultiIndex**(*NpyIter* iter*, char** *errmsg*)

Returns a function pointer for getting the current multi-index of the iterator. Returns NULL if the iterator is not tracking a multi-index. It is recommended that this function pointer be cached in a local variable before the iteration loop.

Returns NULL if there is an error. If errmsg is non-NULL, no Python exception is set when `NPY_FAIL` is returned. Instead, *errmsg is set to an error message. When errmsg is non-NULL, the function may be safely called without holding the Python GIL.

char** **NpyIter_GetDataPtrArray**(*NpyIter* iter*)

This gives back a pointer to the `nop` data pointers. If *`NPY_ITER_EXTERNAL_LOOP`* was not specified, each data pointer points to the current data item of the iterator. If no inner iteration was specified, it points to the first data item of the inner loop.

This pointer may be cached before the iteration loop, calling `iternext` will not change it. This function may be safely called without holding the Python GIL.

char** **NpyIter_GetInitialDataPtrArray**(*NpyIter* iter*)

Gets the array of data pointers directly into the arrays (never into the buffers), corresponding to iteration index 0.

These pointers are different from the pointers accepted by `NpyIter_ResetBasePointers`, because the direction along some axes may have been reversed.

This function may be safely called without holding the Python GIL.

*npy_intp** **NpyIter_GetIndexPtr**(*NpyIter* iter*)

This gives back a pointer to the index being tracked, or NULL if no index is being tracked. It is only useable if one of the flags *`NPY_ITER_C_INDEX`* or *`NPY_ITER_F_INDEX`* were specified during construction.

When the flag *`NPY_ITER_EXTERNAL_LOOP`* is used, the code needs to know the parameters for doing the inner loop. These functions provide that information.

*npy_intp** **`NpyIter_GetInnerStrideArray`**(*NpyIter** *iter*)

Returns a pointer to an array of the `nop` strides, one for each iterated object, to be used by the inner loop.

This pointer may be cached before the iteration loop, calling `iternext` will not change it. This function may be safely called without holding the Python GIL.

**WARNING**: While the pointer may be cached, its values may change if the iterator is buffered.

*npy_intp** **`NpyIter_GetInnerLoopSizePtr`**(*NpyIter** *iter*)

Returns a pointer to the number of iterations the inner loop should execute.

This address may be cached before the iteration loop, calling `iternext` will not change it. The value itself may change during iteration, in particular if buffering is enabled. This function may be safely called without holding the Python GIL.

void **`NpyIter_GetInnerFixedStrideArray`**(*NpyIter** *iter*, *npy_intp** *out_strides*)

Gets an array of strides which are fixed, or will not change during the entire iteration. For strides that may change, the value NPY_MAX_INTP is placed in the stride.

Once the iterator is prepared for iteration (after a reset if `NPY_DELAY_BUFALLOC` was used), call this to get the strides which may be used to select a fast inner loop function. For example, if the stride is 0, that means the inner loop can always load its value into a variable once, then use the variable throughout the loop, or if the stride equals the itemsize, a contiguous version for that operand may be used.

This function may be safely called without holding the Python GIL.

### 8.5.7 Converting from Previous NumPy Iterators

The old iterator API includes functions like PyArrayIter_Check, PyArray_Iter* and PyArray_ITER_*. The multi-iterator array includes PyArray_MultiIter*, PyArray_Broadcast, and PyArray_RemoveSmallest. The new iterator design replaces all of this functionality with a single object and associated API. One goal of the new API is that all uses of the existing iterator should be replaceable with the new iterator without significant effort. In 1.6, the major exception to this is the neighborhood iterator, which does not have corresponding features in this iterator.

Here is a conversion table for which functions to use with the new iterator:

| *Iterator Functions* | |
|---|---|
| `PyArray_IterNew` | `NpyIter_New` |
| `PyArray_IterAllButAxis` | `NpyIter_New` + `axes` parameter **or** Iterator flag `NPY_ITER_EXTERNAL_LOOP` |
| `PyArray_BroadcastToShape` | **NOT SUPPORTED** (Use the support for multiple operands instead.) |
| `PyArrayIter_Check` | Will need to add this in Python exposure |
| `PyArray_ITER_RESET` | `NpyIter_Reset` |
| `PyArray_ITER_NEXT` | Function pointer from `NpyIter_GetIterNext` |
| `PyArray_ITER_DATA` | `NpyIter_GetDataPtrArray` |
| `PyArray_ITER_GOTO` | `NpyIter_GotoMultiIndex` |
| `PyArray_ITER_GOTO1D` | `NpyIter_GotoIndex` or `NpyIter_GotoIterIndex` |
| `PyArray_ITER_NOTDONE` | Return value of `iternext` function pointer |
| *Multi-iterator Functions* | |
| `PyArray_MultiIterNew` | `NpyIter_MultiNew` |
| `PyArray_MultiIter_RESET` | `NpyIter_Reset` |
| `PyArray_MultiIter_NEXT` | Function pointer from `NpyIter_GetIterNext` |
| `PyArray_MultiIter_DATA` | `NpyIter_GetDataPtrArray` |
| `PyArray_MultiIter_NEXTi` | **NOT SUPPORTED** (always lock-step iteration) |
| `PyArray_MultiIter_GOTO` | `NpyIter_GotoMultiIndex` |
| `PyArray_MultiIter_GOTO1D` | `NpyIter_GotoIndex` or `NpyIter_GotoIterIndex` |
| `PyArray_MultiIter_NOTDONE` | Return value of `iternext` function pointer |
| `PyArray_Broadcast` | Handled by `NpyIter_MultiNew` |
| `PyArray_RemoveSmallest` | Iterator flag `NPY_ITER_EXTERNAL_LOOP` |
| *Other Functions* | |
| `PyArray_ConvertToCommonType` | Iterator flag `NPY_ITER_COMMON_DTYPE` |

## 8.6 UFunc API

### 8.6.1 Constants

**`UFUNC_ERR_{HANDLER}`**

`{HANDLER}` can be **IGNORE**, **WARN**, **RAISE**, or **CALL**

**`UFUNC_{THING}_{ERR}`**

`{THING}` can be **MASK**, **SHIFT**, or **FPE**, and `{ERR}` can be **DIVIDEBYZERO**, **OVERFLOW**, **UNDERFLOW**, and **INVALID**.

**`PyUFunc_{VALUE}`**

**`PyUFunc_One`**

**`PyUFunc_Zero`**

**`PyUFunc_MinusOne`**

**`PyUFunc_ReorderableNone`**

**`PyUFunc_None`**

**`PyUFunc_IdentityValue`**

## 8.6.2 Macros

**`NPY_LOOP_BEGIN_THREADS`**

Used in universal function code to only release the Python GIL if loop->obj is not true (*i.e.* this is not an OBJECT array loop). Requires use of `NPY_BEGIN_THREADS_DEF` in variable declaration area.

**`NPY_LOOP_END_THREADS`**

Used in universal function code to re-acquire the Python GIL if it was released (because loop->obj was not true).

## 8.6.3 Functions

PyObject* **`PyUFunc_FromFuncAndData`** (PyUFuncGenericFunction* *func*, void** *data*, char* *types*, int *ntypes*, int *nin*, int *nout*, int *identity*, char* *name*, char* *doc*, int *unused*)

Create a new broadcasting universal function from required variables. Each ufunc builds around the notion of an element-by-element operation. Each ufunc object contains pointers to 1-d loops implementing the basic functionality for each supported type.

**Note:** The *func*, *data*, *types*, *name*, and *doc* arguments are not copied by `PyUFunc_FromFuncAndData`. The caller must ensure that the memory used by these arrays is not freed as long as the ufunc object is alive.

**Parameters**

- **`func`** – Must to an array of length *ntypes* containing `PyUFuncGenericFunction` items. These items are pointers to functions that actually implement the underlying (element-by-element) function $N$ times with the following signature:

  **`void loopfunc(`**
  **`char** args, npy_intp const *dimensions, npy_intp const *steps, void* data)`**

  *args*
  An array of pointers to the actual data for the input and output arrays. The input arguments are given first followed by the output arguments.

  *dimensions*
  A pointer to the size of the dimension over which this function is looping.

  *steps*
  A pointer to the number of bytes to jump to get to the next element in this dimension for each of the input and output arguments.

  *data*
  Arbitrary data (extra arguments, function names, *etc.* ) that can be stored with the ufunc and will be passed in when it is called.

  This is an example of a func specialized for addition of doubles returning doubles.

```
static void
double_add(char **args,
           npy_intp const *dimensions,
           npy_intp const *steps,
           void *extra)
{
    npy_intp i;
    npy_intp is1 = steps[0], is2 = steps[1];
    npy_intp os = steps[2], n = dimensions[0];
    char *i1 = args[0], *i2 = args[1], *op = args[2];
    for (i = 0; i < n; i++) {
```

(continues on next page)

(continued from previous page)

```
            *((double *)op) = *((double *)i1) +
                              *((double *)i2);
            i1 += is1;
            i2 += is2;
            op += os;
        }
}
```

- **`data`** – Should be `NULL` or a pointer to an array of size *ntypes* . This array may contain arbitrary extra-data to be passed to the corresponding loop function in the func array.
- **`types`** – Length `(nin + nout) * ntypes` array of `char` encoding the *`numpy.dtype.num`* (built-in only) that the corresponding function in the `func` array accepts. For instance, for a comparison ufunc with three `ntypes`, two `nin` and one `nout`, where the first function accepts `numpy.int32` and the the second `numpy.int64`, with both returning `numpy.bool_`, `types` would be `(char[]) {5, 5, 0, 7, 7, 0}` since `NPY_INT32` is 5, `NPY_INT64` is 7, and `NPY_BOOL` is 0.

  The bit-width names can also be used (e.g. *`NPY_INT32`*, *`NPY_COMPLEX128`* ) if desired.

  *Casting Rules* will be used at runtime to find the first `func` callable by the input/output provided.
- **`ntypes`** – How many different data-type-specific functions the ufunc has implemented.
- **`nin`** – The number of inputs to this operation.
- **`nout`** – The number of outputs
- **`identity`** – Either *`PyUFunc_One`*, *`PyUFunc_Zero`*, *`PyUFunc_MinusOne`*, or *`PyUFunc_None`*. This specifies what should be returned when an empty array is passed to the reduce method of the ufunc. The special value *`PyUFunc_IdentityValue`* may only be used with the *`PyUFunc_FromFuncAndDataAndSignatureAndIdentity`* method, to allow an arbitrary python object to be used as the identity.
- **`name`** – The name for the ufunc as a `NULL` terminated string. Specifying a name of 'add' or 'multiply' enables a special behavior for integer-typed reductions when no dtype is given. If the input type is an integer (or boolean) data type smaller than the size of the `numpy.int_` data type, it will be internally upcast to the `numpy.int_` (or `numpy.uint`) data type.
- **`doc`** – Allows passing in a documentation string to be stored with the ufunc. The documentation string should not contain the name of the function or the calling signature as that will be dynamically determined from the object and available when accessing the **__doc__** attribute of the ufunc.
- **`unused`** – Unused and present for backwards compatibility of the C-API.

PyObject* **`PyUFunc_FromFuncAndDataAndSignature`** (PyUFuncGenericFunction* *func*, void** *data*, char* *types*, int *ntypes*, int *nin*, int *nout*, int *identity*, char* *name*, char* *doc*, int *unused*, char **signature*)

This function is very similar to PyUFunc_FromFuncAndData above, but has an extra *signature* argument, to define a *generalized universal functions*. Similarly to how ufuncs are built around an element-by-element operation, gufuncs are around subarray-by-subarray operations, the *signature* defining the subarrays to operate on.

**Parameters**

- **`signature`** – The signature for the new gufunc. Setting it to NULL is equivalent to calling PyUFunc_FromFuncAndData. A copy of the string is made, so the passed in buffer can be freed.

PyObject* **`PyUFunc_FromFuncAndDataAndSignatureAndIdentity`**(PyUFuncGenericFunction **func*, void ***data*, char **types*, int *ntypes*, int *nin*, int *nout*, int *identity*, char **name*, char **doc*, int *unused*, char **signature*, PyObject **identity_value*)

This function is very similar to *PyUFunc_FromFuncAndDataAndSignature* above, but has an extra *identity_value* argument, to define an arbitrary identity for the ufunc when `identity` is passed as `PyUFunc_IdentityValue`.

**Parameters**

- **`identity_value`** – The identity for the new gufunc. Must be passed as `NULL` unless the `identity` argument is `PyUFunc_IdentityValue`. Setting it to NULL is equivalent to calling PyUFunc_FromFuncAndDataAndSignature.

int **`PyUFunc_RegisterLoopForType`**(*PyUFuncObject** *ufunc*, int *usertype*, PyUFuncGenericFunction *function*, int* *arg_types*, void* *data*)

This function allows the user to register a 1-d loop with an already- created ufunc to be used whenever the ufunc is called with any of its input arguments as the user-defined data-type. This is needed in order to make ufuncs work with built-in data-types. The data-type must have been previously registered with the numpy system. The loop is passed in as *function*. This loop can take arbitrary data which should be passed in as *data*. The data-types the loop requires are passed in as *arg_types* which must be a pointer to memory at least as large as ufunc->nargs.

int **`PyUFunc_RegisterLoopForDescr`**(*PyUFuncObject** *ufunc*, *PyArray_Descr** *userdtype*, PyUFuncGenericFunction *function*, *PyArray_Descr*** *arg_dtypes*, void* *data*)

This function behaves like PyUFunc_RegisterLoopForType above, except that it allows the user to register a 1-d loop using PyArray_Descr objects instead of dtype type num values. This allows a 1-d loop to be registered for structured array data-dtypes and custom data-types instead of scalar data-types.

int **`PyUFunc_ReplaceLoopBySignature`**(*PyUFuncObject** *ufunc*, PyUFuncGenericFunction *newfunc*, int* *signature*, PyUFuncGenericFunction* *oldfunc*)

Replace a 1-d loop matching the given *signature* in the already-created *ufunc* with the new 1-d loop newfunc. Return the old 1-d loop function in *oldfunc*. Return 0 on success and -1 on failure. This function works only with built-in types (use *`PyUFunc_RegisterLoopForType`* for user-defined types). A signature is an array of data-type numbers indicating the inputs followed by the outputs assumed by the 1-d loop.

int **`PyUFunc_GenericFunction`**(*PyUFuncObject** *self*, PyObject* *args*, PyObject* *kwds*, *PyArrayObject*** *mps*)

Deprecated since version NumPy: 1.19

Unless NumPy is made aware of an issue with this, this function is scheduled for rapid removal without replacement.

Instead of this function `PyObject_Call(ufunc, args, kwds)` should be used. The above function differs from this because it ignores support for non-array, or array subclasses as inputs. To ensure identical behaviour, it may be necessary to convert all inputs using `PyArray_FromAny(obj, NULL, 0, 0, NPY_ARRAY_ENSUREARRAY, NULL)`.

int **`PyUFunc_checkfperr`**(int *errmask*, PyObject* *errobj*)

A simple interface to the IEEE error-flag checking support. The *errmask* argument is a mask of `UFUNC_MASK_{ERR}` bitmasks indicating which errors to check for (and how to check for them). The *errobj* must be a Python tuple with two elements: a string containing the name which will be used in any communication of error and either a callable Python object (call-back function) or `Py_None`. The callable object will only be used if `UFUNC_ERR_CALL` is set as the desired error checking method. This routine manages the GIL and is safe to call even after releasing the GIL. If an error in the IEEE-compatible hardware is determined a -1 is returned, otherwise a 0 is returned.

void **PyUFunc_clearfperr**()

Clear the IEEE error flags.

void **PyUFunc_GetPyValues**(char* *name*, int* *bufsize*, int* *errmask*, PyObject** *errobj*)

Get the Python values used for ufunc processing from the thread-local storage area unless the defaults have been set in which case the name lookup is bypassed. The name is placed as a string in the first element of **errobj*. The second element is the looked-up function to call on error callback. The value of the looked-up buffer-size to use is passed into *bufsize*, and the value of the error mask is placed into *errmask*.

### 8.6.4 Generic functions

At the core of every ufunc is a collection of type-specific functions that defines the basic functionality for each of the supported types. These functions must evaluate the underlying function $N \geq 1$ times. Extra-data may be passed in that may be used during the calculation. This feature allows some general functions to be used as these basic looping functions. The general function has all the code needed to point variables to the right place and set up a function call. The general function assumes that the actual function to call is passed in as the extra data and calls it with the correct values. All of these functions are suitable for placing directly in the array of functions stored in the functions member of the PyUFuncObject structure.

void **PyUFunc_f_f_As_d_d**(char** *args*, npy_intp const **dimensions*, npy_intp const **steps*, void* *func*)

void **PyUFunc_d_d**(char** *args*, npy_intp const **dimensions*, npy_intp const **steps*, void* *func*)

void **PyUFunc_f_f**(char** *args*, npy_intp const **dimensions*, npy_intp const **steps*, void* *func*)

void **PyUFunc_g_g**(char** *args*, npy_intp const **dimensions*, npy_intp const **steps*, void* *func*)

void **PyUFunc_F_F_As_D_D**(char** *args*, npy_intp const **dimensions*, npy_intp const **steps*, void* *func*)

void **PyUFunc_F_F**(char** *args*, npy_intp const **dimensions*, npy_intp const **steps*, void* *func*)

void **PyUFunc_D_D**(char** *args*, npy_intp const **dimensions*, npy_intp const **steps*, void* *func*)

void **PyUFunc_G_G**(char** *args*, npy_intp const **dimensions*, npy_intp const **steps*, void* *func*)

void **PyUFunc_e_e**(char** *args*, npy_intp const **dimensions*, npy_intp const **steps*, void* *func*)

void **PyUFunc_e_e_As_f_f**(char** *args*, npy_intp const **dimensions*, npy_intp const **steps*, void* *func*)

void **PyUFunc_e_e_As_d_d**(char** *args*, npy_intp const **dimensions*, npy_intp const **steps*, void* *func*)

Type specific, core 1-d functions for ufuncs where each calculation is obtained by calling a function taking one input argument and returning one output. This function is passed in `func`. The letters correspond to dtypechar's of the supported data types ( `e` - half, `f` - float, `d` - double, `g` - long double, `F` - cfloat, `D` - cdouble, `G` - clongdouble). The argument *func* must support the same signature. The _As_X_X variants assume ndarray's of one data type but cast the values to use an underlying function that takes a different data type. Thus, `PyUFunc_f_f_As_d_d` uses ndarrays of data type `NPY_FLOAT` but calls out to a C-function that takes double and returns double.

void **PyUFunc_ff_f_As_dd_d**(char** *args*, npy_intp const **dimensions*, npy_intp const **steps*, void* *func*)

void **PyUFunc_ff_f**(char** *args*, npy_intp const **dimensions*, npy_intp const **steps*, void* *func*)

void **PyUFunc_dd_d**(char** *args*, npy_intp const **dimensions*, npy_intp const **steps*, void* *func*)

void **PyUFunc_gg_g**(char** *args*, npy_intp const **dimensions*, npy_intp const **steps*, void* *func*)

void **PyUFunc_FF_F_As_DD_D**(char** *args*, npy_intp const **dimensions*, npy_intp const **steps*, void* *func*)

void **PyUFunc_DD_D**(char** *args*, npy_intp const **dimensions*, npy_intp const **steps*, void* *func*)

void **PyUFunc_FF_F**(char** *args*, npy_intp const **dimensions*, npy_intp const **steps*, void* *func*) void **PyUFunc_GG_G**(char** *args*, npy_intp const **dimensions*, npy_intp const **steps*, void* *func*)

void **PyUFunc_ee_e**(char** *args*, npy_intp const **dimensions*, npy_intp const **steps*, void* *func*)

void **PyUFunc_ee_e_As_ff_f**(char** *args*, npy_intp const **dimensions*, npy_intp const **steps*, void* *func*)

void **PyUFunc_ee_e_As_dd_d**(char** *args*, npy_intp const **dimensions*, npy_intp const **steps*, void* *func*)

Type specific, core 1-d functions for ufuncs where each calculation is obtained by calling a function taking two input arguments and returning one output. The underlying function to call is passed in as *func*. The letters correspond to dtypechar's of the specific data type supported by the general-purpose function. The argument `func` must support the corresponding signature. The `_As_XX_X` variants assume ndarrays of one data type but cast the values at each iteration of the loop to use the underlying function that takes a different data type.

void **PyUFunc_O_O**(char** *args*, npy_intp const **dimensions*, npy_intp const **steps*, void* *func*)

void **PyUFunc_OO_O**(char** *args*, npy_intp const **dimensions*, npy_intp const **steps*, void* *func*)

One-input, one-output, and two-input, one-output core 1-d functions for the `NPY_OBJECT` data type. These functions handle reference count issues and return early on error. The actual function to call is *func* and it must accept calls with the signature `(PyObject*) (PyObject*)` for `PyUFunc_O_O` or `(PyObject*)(PyObject *, PyObject *)` for `PyUFunc_OO_O`.

void **PyUFunc_O_O_method**(char** *args*, npy_intp const **dimensions*, npy_intp const **steps*, void* *func*)

This general purpose 1-d core function assumes that *func* is a string representing a method of the input object. For each iteration of the loop, the Python object is extracted from the array and its *func* method is called returning the result to the output array.

void **PyUFunc_OO_O_method**(char** *args*, npy_intp const **dimensions*, npy_intp const **steps*, void* *func*)

This general purpose 1-d core function assumes that *func* is a string representing a method of the input object that takes one argument. The first argument in *args* is the method whose function is called, the second argument in *args* is the argument passed to the function. The output of the function is stored in the third entry of *args*.

void **PyUFunc_On_Om**(char** *args*, npy_intp const **dimensions*, npy_intp const **steps*, void* *func*)

This is the 1-d core function used by the dynamic ufuncs created by umath.frompyfunc(function, nin, nout). In this case *func* is a pointer to a `PyUFunc_PyFuncData` structure which has definition

**PyUFunc_PyFuncData**

```
typedef struct {
    int nin;
    int nout;
    PyObject *callable;
} PyUFunc_PyFuncData;
```

At each iteration of the loop, the *nin* input objects are extracted from their object arrays and placed into an argument tuple, the Python *callable* is called with the input arguments, and the nout outputs are placed into their object arrays.

### 8.6.5 Importing the API

**PY_UFUNC_UNIQUE_SYMBOL**

**NO_IMPORT_UFUNC**

void **import_ufunc**(void)

These are the constants and functions for accessing the ufunc C-API from extension modules in precisely the same way as the array C-API can be accessed. The `import_ufunc ()` function must always be called (in the initialization subroutine of the extension module). If your extension module is in one file then that is all that is required. The other two constants are useful if your extension module makes use of multiple files. In that case, define `PY_UFUNC_UNIQUE_SYMBOL` to something unique to your code and then in source files that do not contain the module initialization function but still need access to the UFUNC API, define *PY_UFUNC_UNIQUE_SYMBOL* to the same name used previously and also define *NO_IMPORT_UFUNC*.

The C-API is actually an array of function pointers. This array is created (and pointed to by a global variable) by import_ufunc. The global variable is either statically defined or allowed to be seen by other files depending on the state of *PY_UFUNC_UNIQUE_SYMBOL* and *NO_IMPORT_UFUNC*.

## 8.7 Generalized Universal Function API

There is a general need for looping over not only functions on scalars but also over functions on vectors (or arrays). This concept is realized in NumPy by generalizing the universal functions (ufuncs). In regular ufuncs, the elementary function is limited to element-by-element operations, whereas the generalized version (gufuncs) supports "sub-array" by "sub-array" operations. The Perl vector library PDL provides a similar functionality and its terms are re-used in the following.

Each generalized ufunc has information associated with it that states what the "core" dimensionality of the inputs is, as well as the corresponding dimensionality of the outputs (the element-wise ufuncs have zero core dimensions). The list of the core dimensions for all arguments is called the "signature" of a ufunc. For example, the ufunc numpy.add has signature `(),()->()` defining two scalar inputs and one scalar output.

Another example is the function `inner1d(a, b)` with a signature of `(i),(i)->()`. This applies the inner product along the last axis of each input, but keeps the remaining indices intact. For example, where `a` is of shape `(3, 5, N)` and `b` is of shape `(5, N)`, this will return an output of shape `(3,5)`. The underlying elementary function is called `3 * 5` times. In the signature, we specify one core dimension `(i)` for each input and zero core dimensions `()` for the output, since it takes two 1-d arrays and returns a scalar. By using the same name `i`, we specify that the two corresponding dimensions should be of the same size.

The dimensions beyond the core dimensions are called "loop" dimensions. In the above example, this corresponds to `(3, 5)`.

The signature determines how the dimensions of each input/output array are split into core and loop dimensions:

1. Each dimension in the signature is matched to a dimension of the corresponding passed-in array, starting from the end of the shape tuple. These are the core dimensions, and they must be present in the arrays, or an error will be raised.
2. Core dimensions assigned to the same label in the signature (e.g. the `i` in `inner1d`'s `(i),(i)->()`) must have exactly matching sizes, no broadcasting is performed.
3. The core dimensions are removed from all inputs and the remaining dimensions are broadcast together, defining the loop dimensions.
4. The shape of each output is determined from the loop dimensions plus the output's core dimensions

Typically, the size of all core dimensions in an output will be determined by the size of a core dimension with the same label in an input array. This is not a requirement, and it is possible to define a signature where a label comes up for the first time in an output, although some precautions must be taken when calling such a function. An example would be the function `euclidean_pdist(a)`, with signature `(n,d)->(p)`, that given an array of `n` `d`-dimensional vectors, computes all unique pairwise Euclidean distances among them. The output dimension `p` must therefore be equal to `n * (n - 1) / 2`, but it is the caller's responsibility to pass in an output array of the right size. If the size of a core dimension of an output cannot be determined from a passed in input or output array, an error will be raised.

Note: Prior to NumPy 1.10.0, less strict checks were in place: missing core dimensions were created by prepending 1's to the shape as necessary, core dimensions with the same label were broadcast together, and undetermined dimensions were created with size 1.

## 8.7.1 Definitions

**Elementary Function**

Each ufunc consists of an elementary function that performs the most basic operation on the smallest portion of array arguments (e.g. adding two numbers is the most basic operation in adding two arrays). The ufunc applies the elementary function multiple times on different parts of the arrays. The input/output of elementary functions can be vectors; e.g., the elementary function of inner1d takes two vectors as input.

**Signature**

A signature is a string describing the input/output dimensions of the elementary function of a ufunc. See section below for more details.

**Core Dimension**

The dimensionality of each input/output of an elementary function is defined by its core dimensions (zero core dimensions correspond to a scalar input/output). The core dimensions are mapped to the last dimensions of the input/output arrays.

**Dimension Name**

A dimension name represents a core dimension in the signature. Different dimensions may share a name, indicating that they are of the same size.

**Dimension Index**

A dimension index is an integer representing a dimension name. It enumerates the dimension names according to the order of the first occurrence of each name in the signature.

## 8.7.2 Details of Signature

The signature defines "core" dimensionality of input and output variables, and thereby also defines the contraction of the dimensions. The signature is represented by a string of the following format:

- Core dimensions of each input or output array are represented by a list of dimension names in parentheses, `(i_1,...,i_N)`; a scalar input/output is denoted by `()`. Instead of `i_1`, `i_2`, etc, one can use any valid Python variable name.
- Dimension lists for different arguments are separated by `","`. Input/output arguments are separated by `"->"`.
- If one uses the same dimension name in multiple locations, this enforces the same size of the corresponding dimensions.

The formal syntax of signatures is as follows:

```
<Signature>            ::= <Input arguments> "->" <Output arguments>
<Input arguments>      ::= <Argument list>
<Output arguments>     ::= <Argument list>
<Argument list>        ::= nil | <Argument> | <Argument> "," <Argument list>
<Argument>             ::= "(" <Core dimension list> ")"
<Core dimension list>  ::= nil | <Core dimension> |
                           <Core dimension> "," <Core dimension list>
<Core dimension>       ::= <Dimension name> <Dimension modifier>
<Dimension name>       ::= valid Python variable name | valid integer
<Dimension modifier>   ::= nil | "?"
```

Notes:

1. All quotes are for clarity.

2. Unmodified core dimensions that share the same name must have the same size. Each dimension name typically corresponds to one level of looping in the elementary function's implementation.
3. White spaces are ignored.
4. An integer as a dimension name freezes that dimension to the value.
5. If the name is suffixed with the "?" modifier, the dimension is a core dimension only if it exists on all inputs and outputs that share it; otherwise it is ignored (and replaced by a dimension of size 1 for the elementary function).

Here are some examples of signatures:

| name | signature | common usage |
|---|---|---|
| add | `(),()->()` | binary ufunc |
| sum1d | `(i)->()` | reduction |
| inner1d | `(i),(i)->()` | vector-vector multiplication |
| matmat | `(m,n),(n,p)->(m,p)` | matrix multiplication |
| vecmat | `(n),(n,p)->(p)` | vector-matrix multiplication |
| matvec | `(m,n),(n)->(m)` | matrix-vector multiplication |
| matmul | `(m?,n),(n,p?)->(m?,p?)` | combination of the four above |
| outer_inner | `(i,t),(j,t)->(i,j)` | inner over the last dimension, outer over the second to last, and loop/broadcast over the rest. |
| cross1d | `(3),(3)->(3)` | cross product where the last dimension is frozen and must be 3 |

The last is an instance of freezing a core dimension and can be used to improve ufunc performance

### 8.7.3 C-API for implementing Elementary Functions

The current interface remains unchanged, and `PyUFunc_FromFuncAndData` can still be used to implement (specialized) ufuncs, consisting of scalar elementary functions.

One can use `PyUFunc_FromFuncAndDataAndSignature` to declare a more general ufunc. The argument list is the same as `PyUFunc_FromFuncAndData`, with an additional argument specifying the signature as C string.

Furthermore, the callback function is of the same type as before, `void (*foo)(char **args, intp *dimensions, intp *steps, void *func)`. When invoked, `args` is a list of length `nargs` containing the data of all input/output arguments. For a scalar elementary function, `steps` is also of length `nargs`, denoting the strides used for the arguments. `dimensions` is a pointer to a single integer defining the size of the axis to be looped over.

For a non-trivial signature, `dimensions` will also contain the sizes of the core dimensions as well, starting at the second entry. Only one size is provided for each unique dimension name and the sizes are given according to the first occurrence of a dimension name in the signature.

The first `nargs` elements of `steps` remain the same as for scalar ufuncs. The following elements contain the strides of all core dimensions for all arguments in order.

For example, consider a ufunc with signature `(i,j),(i)->()`. In this case, `args` will contain three pointers to the data of the input/output arrays `a`, `b`, `c`. Furthermore, `dimensions` will be `[N, I, J]` to define the size of `N` of the loop and the sizes `I` and `J` for the core dimensions `i` and `j`. Finally, `steps` will be `[a_N, b_N, c_N, a_i, a_j, b_i]`, containing all necessary strides.

## 8.8 NumPy core libraries

New in version 1.3.0.

Starting from numpy 1.3.0, we are working on separating the pure C, "computational" code from the python dependent code. The goal is twofolds: making the code cleaner, and enabling code reuse by other extensions outside numpy (scipy, etc…).

### 8.8.1 NumPy core math library

The numpy core math library ('npymath') is a first step in this direction. This library contains most math-related C99 functionality, which can be used on platforms where C99 is not well supported. The core math functions have the same API as the C99 ones, except for the npy_* prefix.

The available functions are defined in <numpy/npy_math.h> - please refer to this header when in doubt.

#### Floating point classification

**`NPY_NAN`**

This macro is defined to a NaN (Not a Number), and is guaranteed to have the signbit unset ('positive' NaN). The corresponding single and extension precision macro are available with the suffix F and L.

**`NPY_INFINITY`**

This macro is defined to a positive inf. The corresponding single and extension precision macro are available with the suffix F and L.

**`NPY_PZERO`**

This macro is defined to positive zero. The corresponding single and extension precision macro are available with the suffix F and L.

**`NPY_NZERO`**

This macro is defined to negative zero (that is with the sign bit set). The corresponding single and extension precision macro are available with the suffix F and L.

int **`npy_isnan`**`(x)`

This is a macro, and is equivalent to C99 isnan: works for single, double and extended precision, and return a non 0 value is x is a NaN.

int **`npy_isfinite`**`(x)`

This is a macro, and is equivalent to C99 isfinite: works for single, double and extended precision, and return a non 0 value is x is neither a NaN nor an infinity.

int **`npy_isinf`**`(x)`

This is a macro, and is equivalent to C99 isinf: works for single, double and extended precision, and return a non 0 value is x is infinite (positive and negative).

int **`npy_signbit`**`(x)`

This is a macro, and is equivalent to C99 signbit: works for single, double and extended precision, and return a non 0 value is x has the signbit set (that is the number is negative).

double **`npy_copysign`**(double *x*, double *y*)

This is a function equivalent to C99 copysign: return x with the same sign as y. Works for any value, including inf and nan. Single and extended precisions are available with suffix f and l.

New in version 1.4.0.

### Useful math constants

The following math constants are available in `npy_math.h`. Single and extended precision are also available by adding the `f` and `l` suffixes respectively.

**NPY_E**
Base of natural logarithm ($e$)

**NPY_LOG2E**
Logarithm to base 2 of the Euler constant ($\frac{\ln(e)}{\ln(2)}$)

**NPY_LOG10E**
Logarithm to base 10 of the Euler constant ($\frac{\ln(e)}{\ln(10)}$)

**NPY_LOGE2**
Natural logarithm of 2 ($\ln(2)$)

**NPY_LOGE10**
Natural logarithm of 10 ($\ln(10)$)

**NPY_PI**
Pi ($\pi$)

**NPY_PI_2**
Pi divided by 2 ($\frac{\pi}{2}$)

**NPY_PI_4**
Pi divided by 4 ($\frac{\pi}{4}$)

**NPY_1_PI**
Reciprocal of pi ($\frac{1}{\pi}$)

**NPY_2_PI**
Two times the reciprocal of pi ($\frac{2}{\pi}$)

**NPY_EULER**

**The Euler constant**

$\lim_{n\rightarrow\infty}(\sum_{k=1}^n \frac{1}{k} - \ln n)$

### Low-level floating point manipulation

Those can be useful for precise floating point comparison.

double **npy_nextafter** (double *x*, double *y*)
This is a function equivalent to C99 nextafter: return next representable floating point value from x in the direction of y. Single and extended precisions are available with suffix f and l.

New in version 1.4.0.

double **npy_spacing** (double *x*)
This is a function equivalent to Fortran intrinsic. Return distance between x and next representable floating point value from x, e.g. spacing(1) == eps. spacing of nan and +/- inf return nan. Single and extended precisions are available with suffix f and l.

New in version 1.4.0.

void **npy_set_floatstatus_divbyzero** ()
Set the divide by zero floating point exception

New in version 1.6.0.

void **`npy_set_floatstatus_overflow`**`()`

Set the overflow floating point exception

New in version 1.6.0.

void **`npy_set_floatstatus_underflow`**`()`

Set the underflow floating point exception

New in version 1.6.0.

void **`npy_set_floatstatus_invalid`**`()`

Set the invalid floating point exception

New in version 1.6.0.

int **`npy_get_floatstatus`**`()`

Get floating point status. Returns a bitmask with following possible flags:

- NPY_FPE_DIVIDEBYZERO
- NPY_FPE_OVERFLOW
- NPY_FPE_UNDERFLOW
- NPY_FPE_INVALID

Note that *`npy_get_floatstatus_barrier`* is preferable as it prevents aggressive compiler optimizations reordering the call relative to the code setting the status, which could lead to incorrect results.

New in version 1.9.0.

int **`npy_get_floatstatus_barrier`**`(`char*`)`

Get floating point status. A pointer to a local variable is passed in to prevent aggressive compiler optimizations from reordering this function call relative to the code setting the status, which could lead to incorrect results.

Returns a bitmask with following possible flags:

- NPY_FPE_DIVIDEBYZERO
- NPY_FPE_OVERFLOW
- NPY_FPE_UNDERFLOW
- NPY_FPE_INVALID

New in version 1.15.0.

int **`npy_clear_floatstatus`**`()`

Clears the floating point status. Returns the previous status mask.

Note that *`npy_clear_floatstatus_barrier`* is preferable as it prevents aggressive compiler optimizations reordering the call relative to the code setting the status, which could lead to incorrect results.

New in version 1.9.0.

int **`npy_clear_floatstatus_barrier`**`(`char*`)`

Clears the floating point status. A pointer to a local variable is passed in to prevent aggressive compiler optimizations from reordering this function call. Returns the previous status mask.

New in version 1.15.0.

### Complex functions

New in version 1.4.0.

C99-like complex functions have been added. Those can be used if you wish to implement portable C extensions. Since we still support platforms without C99 complex type, you need to restrict to C90-compatible syntax, e.g.:

```
/* a = 1 + 2i \*/
npy_complex a = npy_cpack(1, 2);
npy_complex b;

b = npy_log(a);
```

### Linking against the core math library in an extension

New in version 1.4.0.

To use the core math library in your own extension, you need to add the npymath compile and link options to your extension in your setup.py:

```
>>> from numpy.distutils.misc_util import get_info
>>> info = get_info('npymath')
>>> _ = config.add_extension('foo', sources=['foo.c'], extra_info=info)
```

In other words, the usage of info is exactly the same as when using blas_info and co.

### Half-precision functions

New in version 1.6.0.

The header file <numpy/halffloat.h> provides functions to work with IEEE 754-2008 16-bit floating point values. While this format is not typically used for numerical computations, it is useful for storing values which require floating point but do not need much precision. It can also be used as an educational tool to understand the nature of floating point round-off error.

Like for other types, NumPy includes a typedef npy_half for the 16 bit float. Unlike for most of the other types, you cannot use this as a normal type in C, since it is a typedef for npy_uint16. For example, 1.0 looks like 0x3c00 to C, and if you do an equality comparison between the different signed zeros, you will get -0.0 != 0.0 (0x8000 != 0x0000), which is incorrect.

For these reasons, NumPy provides an API to work with npy_half values accessible by including <numpy/halffloat.h> and linking to 'npymath'. For functions that are not provided directly, such as the arithmetic operations, the preferred method is to convert to float or double and back again, as in the following example.

```
npy_half sum(int n, npy_half *array) {
    float ret = 0;
    while(n--) {
        ret += npy_half_to_float(*array++);
    }
    return npy_float_to_half(ret);
}
```

External Links:

- 754-2008 IEEE Standard for Floating-Point Arithmetic
- Half-precision Float Wikipedia Article.

- OpenGL Half Float Pixel Support
- The OpenEXR image format.

**NPY_HALF_ZERO**
This macro is defined to positive zero.

**NPY_HALF_PZERO**
This macro is defined to positive zero.

**NPY_HALF_NZERO**
This macro is defined to negative zero.

**NPY_HALF_ONE**
This macro is defined to 1.0.

**NPY_HALF_NEGONE**
This macro is defined to -1.0.

**NPY_HALF_PINF**
This macro is defined to +inf.

**NPY_HALF_NINF**
This macro is defined to -inf.

**NPY_HALF_NAN**
This macro is defined to a NaN value, guaranteed to have its sign bit unset.

float **npy_half_to_float** (*npy_half h*)
Converts a half-precision float to a single-precision float.

double **npy_half_to_double** (*npy_half h*)
Converts a half-precision float to a double-precision float.

*npy_half* **npy_float_to_half** (float *f*)
Converts a single-precision float to a half-precision float. The value is rounded to the nearest representable half, with ties going to the nearest even. If the value is too small or too big, the system's floating point underflow or overflow bit will be set.

*npy_half* **npy_double_to_half** (double *d*)
Converts a double-precision float to a half-precision float. The value is rounded to the nearest representable half, with ties going to the nearest even. If the value is too small or too big, the system's floating point underflow or overflow bit will be set.

int **npy_half_eq** (*npy_half h1*, *npy_half h2*)
Compares two half-precision floats (h1 == h2).

int **npy_half_ne** (*npy_half h1*, *npy_half h2*)
Compares two half-precision floats (h1 != h2).

int **npy_half_le** (*npy_half h1*, *npy_half h2*)
Compares two half-precision floats (h1 <= h2).

int **npy_half_lt** (*npy_half h1*, *npy_half h2*)
Compares two half-precision floats (h1 < h2).

int **npy_half_ge** (*npy_half h1*, *npy_half h2*)
Compares two half-precision floats (h1 >= h2).

int **npy_half_gt** (*npy_half h1*, *npy_half h2*)
Compares two half-precision floats (h1 > h2).

int **npy_half_eq_nonan** (*npy_half h1*, *npy_half h2*)

Compares two half-precision floats that are known to not be NaN (h1 == h2). If a value is NaN, the result is undefined.

int **npy_half_lt_nonan** (*npy_half h1*, *npy_half h2*)

Compares two half-precision floats that are known to not be NaN (h1 < h2). If a value is NaN, the result is undefined.

int **npy_half_le_nonan** (*npy_half h1*, *npy_half h2*)

Compares two half-precision floats that are known to not be NaN (h1 <= h2). If a value is NaN, the result is undefined.

int **npy_half_iszero** (*npy_half h*)

Tests whether the half-precision float has a value equal to zero. This may be slightly faster than calling npy_half_eq(h, NPY_ZERO).

int **npy_half_isnan** (*npy_half h*)

Tests whether the half-precision float is a NaN.

int **npy_half_isinf** (*npy_half h*)

Tests whether the half-precision float is plus or minus Inf.

int **npy_half_isfinite** (*npy_half h*)

Tests whether the half-precision float is finite (not NaN or Inf).

int **npy_half_signbit** (*npy_half h*)

Returns 1 is h is negative, 0 otherwise.

*npy_half* **npy_half_copysign** (*npy_half x*, *npy_half y*)

Returns the value of x with the sign bit copied from y. Works for any value, including Inf and NaN.

*npy_half* **npy_half_spacing** (*npy_half h*)

This is the same for half-precision float as npy_spacing and npy_spacingf described in the low-level floating point section.

*npy_half* **npy_half_nextafter** (*npy_half x*, *npy_half y*)

This is the same for half-precision float as npy_nextafter and npy_nextafterf described in the low-level floating point section.

*npy_uint16* **npy_floatbits_to_halfbits** (*npy_uint32 f*)

Low-level function which converts a 32-bit single-precision float, stored as a uint32, into a 16-bit half-precision float.

*npy_uint16* **npy_doublebits_to_halfbits** (*npy_uint64 d*)

Low-level function which converts a 64-bit double-precision float, stored as a uint64, into a 16-bit half-precision float.

*npy_uint32* **npy_halfbits_to_floatbits** (*npy_uint16 h*)

Low-level function which converts a 16-bit half-precision float into a 32-bit single-precision float, stored as a uint32.

*npy_uint64* **npy_halfbits_to_doublebits** (*npy_uint16 h*)

Low-level function which converts a 16-bit half-precision float into a 64-bit double-precision float, stored as a uint64.

# 8.9 C API Deprecations

## 8.9.1 Background

The API exposed by NumPy for third-party extensions has grown over years of releases, and has allowed programmers to directly access NumPy functionality from C. This API can be best described as "organic". It has emerged from multiple competing desires and from multiple points of view over the years, strongly influenced by the desire to make it easy for users to move to NumPy from Numeric and Numarray. The core API originated with Numeric in 1995 and there are patterns such as the heavy use of macros written to mimic Python's C-API as well as account for compiler technology of the late 90's. There is also only a small group of volunteers who have had very little time to spend on improving this API.

There is an ongoing effort to improve the API. It is important in this effort to ensure that code that compiles for NumPy 1.X continues to compile for NumPy 1.X. At the same time, certain API's will be marked as deprecated so that future-looking code can avoid these API's and follow better practices.

Another important role played by deprecation markings in the C API is to move towards hiding internal details of the NumPy implementation. For those needing direct, easy, access to the data of ndarrays, this will not remove this ability. Rather, there are many potential performance optimizations which require changing the implementation details, and NumPy developers have been unable to try them because of the high value of preserving ABI compatibility. By deprecating this direct access, we will in the future be able to improve NumPy's performance in ways we cannot presently.

## 8.9.2 Deprecation Mechanism NPY_NO_DEPRECATED_API

In C, there is no equivalent to the deprecation warnings that Python supports. One way to do deprecations is to flag them in the documentation and release notes, then remove or change the deprecated features in a future major version (NumPy 2.0 and beyond). Minor versions of NumPy should not have major C-API changes, however, that prevent code that worked on a previous minor release. For example, we will do our best to ensure that code that compiled and worked on NumPy 1.4 should continue to work on NumPy 1.7 (but perhaps with compiler warnings).

To use the NPY_NO_DEPRECATED_API mechanism, you need to #define it to the target API version of NumPy before #including any NumPy headers. If you want to confirm that your code is clean against 1.7, use:

```
#define NPY_NO_DEPRECATED_API NPY_1_7_API_VERSION
```

On compilers which support a #warning mechanism, NumPy issues a compiler warning if you do not define the symbol NPY_NO_DEPRECATED_API. This way, the fact that there are deprecations will be flagged for third-party developers who may not have read the release notes closely.

CHAPTER

NINE

# NUMPY INTERNALS

## 9.1 NumPy C Code Explanations

> Fanaticism consists of redoubling your efforts when you have forgotten your aim. — *George Santayana*
>
> An authority is a person who can tell you more about something than you really care to know. — *Unknown*

This Chapter attempts to explain the logic behind some of the new pieces of code. The purpose behind these explanations is to enable somebody to be able to understand the ideas behind the implementation somewhat more easily than just staring at the code. Perhaps in this way, the algorithms can be improved on, borrowed from, and/or optimized by more people.

### 9.1.1 Memory model

One fundamental aspect of the ndarray is that an array is seen as a "chunk" of memory starting at some location. The interpretation of this memory depends on the stride information. For each dimension in an $N$ -dimensional array, an integer (stride) dictates how many bytes must be skipped to get to the next element in that dimension. Unless you have a single-segment array, this stride information must be consulted when traversing through an array. It is not difficult to write code that accepts strides, you just have to use (char *) pointers because strides are in units of bytes. Keep in mind also that strides do not have to be unit-multiples of the element size. Also, remember that if the number of dimensions of the array is 0 (sometimes called a rank-0 array), then the strides and dimensions variables are NULL.

Besides the structural information contained in the strides and dimensions members of the `PyArrayObject`, the flags contain important information about how the data may be accessed. In particular, the `NPY_ARRAY_ALIGNED` flag is set when the memory is on a suitable boundary according to the data-type array. Even if you have a contiguous chunk of memory, you cannot just assume it is safe to dereference a data- type-specific pointer to an element. Only if the `NPY_ARRAY_ALIGNED` flag is set is this a safe operation (on some platforms it will work but on others, like Solaris, it will cause a bus error). The `NPY_ARRAY_WRITEABLE` should also be ensured if you plan on writing to the memory area of the array. It is also possible to obtain a pointer to an unwritable memory area. Sometimes, writing to the memory area when the `NPY_ARRAY_WRITEABLE` flag is not set will just be rude. Other times it can cause program crashes ( *e.g.* a data-area that is a read-only memory-mapped file).

### 9.1.2 Data-type encapsulation

The data-type is an important abstraction of the ndarray. Operations will look to the data-type to provide the key functionality that is needed to operate on the array. This functionality is provided in the list of function pointers pointed to by the 'f' member of the *PyArray_Descr* structure. In this way, the number of data-types can be extended simply by providing a *PyArray_Descr* structure with suitable function pointers in the 'f' member. For built-in types there are some optimizations that by-pass this mechanism, but the point of the data- type abstraction is to allow new data-types to be added.

One of the built-in data-types, the void data-type allows for arbitrary structured types containing 1 or more fields as elements of the array. A field is simply another data-type object along with an offset into the current structured type. In order to support arbitrarily nested fields, several recursive implementations of data-type access are implemented for the void type. A common idiom is to cycle through the elements of the dictionary and perform a specific operation based on the data-type object stored at the given offset. These offsets can be arbitrary numbers. Therefore, the possibility of encountering mis- aligned data must be recognized and taken into account if necessary.

### 9.1.3 N-D Iterators

A very common operation in much of NumPy code is the need to iterate over all the elements of a general, strided, N-dimensional array. This operation of a general-purpose N-dimensional loop is abstracted in the notion of an iterator object. To write an N-dimensional loop, you only have to create an iterator object from an ndarray, work with the dataptr member of the iterator object structure and call the macro *PyArray_ITER_NEXT* (it) on the iterator object to move to the next element. The "next" element is always in C-contiguous order. The macro works by first special casing the C-contiguous, 1-D, and 2-D cases which work very simply.

For the general case, the iteration works by keeping track of a list of coordinate counters in the iterator object. At each iteration, the last coordinate counter is increased (starting from 0). If this counter is smaller than one less than the size of the array in that dimension (a pre-computed and stored value), then the counter is increased and the dataptr member is increased by the strides in that dimension and the macro ends. If the end of a dimension is reached, the counter for the last dimension is reset to zero and the dataptr is moved back to the beginning of that dimension by subtracting the strides value times one less than the number of elements in that dimension (this is also pre-computed and stored in the backstrides member of the iterator object). In this case, the macro does not end, but a local dimension counter is decremented so that the next-to-last dimension replaces the role that the last dimension played and the previously-described tests are executed again on the next-to-last dimension. In this way, the dataptr is adjusted appropriately for arbitrary striding.

The coordinates member of the *PyArrayIterObject* structure maintains the current N-d counter unless the underlying array is C-contiguous in which case the coordinate counting is by-passed. The index member of the *PyArrayIterObject* keeps track of the current flat index of the iterator. It is updated by the *PyArray_ITER_NEXT* macro.

### 9.1.4 Broadcasting

In Numeric, the ancestor of Numpy, broadcasting was implemented in several lines of code buried deep in ufuncobject.c. In NumPy, the notion of broadcasting has been abstracted so that it can be performed in multiple places. Broadcasting is handled by the function *PyArray_Broadcast*. This function requires a *PyArrayMultiIterObject* (or something that is a binary equivalent) to be passed in. The *PyArrayMultiIterObject* keeps track of the broadcast number of dimensions and size in each dimension along with the total size of the broadcast result. It also keeps track of the number of arrays being broadcast and a pointer to an iterator for each of the arrays being broadcast.

The *PyArray_Broadcast* function takes the iterators that have already been defined and uses them to determine the broadcast shape in each dimension (to create the iterators at the same time that broadcasting occurs then use the `PyMultiIter_New` function). Then, the iterators are adjusted so that each iterator thinks it is iterating over an array with the broadcast size. This is done by adjusting the iterators number of dimensions, and the shape in each dimension. This works because the iterator strides are also adjusted. Broadcasting only adjusts (or adds) length-1 dimensions. For these dimensions, the strides variable is simply set to 0 so that the data-pointer for the iterator over that array doesn't move as the broadcasting operation operates over the extended dimension.

Broadcasting was always implemented in Numeric using 0-valued strides for the extended dimensions. It is done in exactly the same way in NumPy. The big difference is that now the array of strides is kept track of in a *`PyArrayIterObject`*, the iterators involved in a broadcast result are kept track of in a *`PyArrayMultiIterObject`*, and the `PyArray_BroadCast` call implements the broad-casting rules.

### 9.1.5 Array Scalars

The array scalars offer a hierarchy of Python types that allow a one- to-one correspondence between the data-type stored in an array and the Python-type that is returned when an element is extracted from the array. An exception to this rule was made with object arrays. Object arrays are heterogeneous collections of arbitrary Python objects. When you select an item from an object array, you get back the original Python object (and not an object array scalar which does exist but is rarely used for practical purposes).

The array scalars also offer the same methods and attributes as arrays with the intent that the same code can be used to support arbitrary dimensions (including 0-dimensions). The array scalars are read-only (immutable) with the exception of the void scalar which can also be written to so that structured array field setting works more naturally (a[0]['f1'] = `value` ).

### 9.1.6 Indexing

All python indexing operations `arr[index]` are organized by first preparing the index and finding the index type. The supported index types are:

- integer
- newaxis
- slice
- ellipsis
- integer arrays/array-likes (fancy)
- boolean (single boolean array); if there is more than one boolean array as index or the shape does not match exactly, the boolean array will be converted to an integer array instead.
- 0-d boolean (and also integer); 0-d boolean arrays are a special case which has to be handled in the advanced indexing code. They signal that a 0-d boolean array had to be interpreted as an integer array.

As well as the scalar array special case signaling that an integer array was interpreted as an integer index, which is important because an integer array index forces a copy but is ignored if a scalar is returned (full integer index). The prepared index is guaranteed to be valid with the exception of out of bound values and broadcasting errors for advanced indexing. This includes that an ellipsis is added for incomplete indices for example when a two dimensional array is indexed with a single integer.

The next step depends on the type of index which was found. If all dimensions are indexed with an integer a scalar is returned or set. A single boolean indexing array will call specialized boolean functions. Indices containing an ellipsis or slice but no advanced indexing will always create a view into the old array by calculating the new strides and memory offset. This view can then either be returned or, for assignments, filled using `PyArray_CopyObject`. Note that *PyArray_CopyObject* may also be called on temporary arrays in other branches to support complicated assignments when the array is of object dtype.

### Advanced indexing

By far the most complex case is advanced indexing, which may or may not be combined with typical view based indexing. Here integer indices are interpreted as view based. Before trying to understand this, you may want to make yourself familiar with its subtleties. The advanced indexing code has three different branches and one special case:

- There is one indexing array and it, as well as the assignment array, can be iterated trivially. For example they may be contiguous. Also the indexing array must be of `intp` type and the value array in assignments should be of the correct type. This is purely a fast path.
- There are only integer array indices so that no subarray exists.
- View based and advanced indexing is mixed. In this case the view based indexing defines a collection of subarrays that are combined by the advanced indexing. For example, `arr[[1, 2, 3], :]` is created by vertically stacking the subarrays `arr[1, :]`, `arr[2,:]`, and `arr[3, :]`.
- There is a subarray but it has exactly one element. This case can be handled as if there is no subarray, but needs some care during setup.

Deciding what case applies, checking broadcasting, and determining the kind of transposition needed are all done in *PyArray_MapIterNew*. After setting up, there are two cases. If there is no subarray or it only has one element, no subarray iteration is necessary and an iterator is prepared which iterates all indexing arrays *as well as* the result or value array. If there is a subarray, there are three iterators prepared. One for the indexing arrays, one for the result or value array (minus its subarray), and one for the subarrays of the original and the result/assignment array. The first two iterators give (or allow calculation) of the pointers into the start of the subarray, which then allows to restart the subarray iteration.

When advanced indices are next to each other transposing may be necessary. All necessary transposing is handled by `PyArray_MapIterSwapAxes` and has to be handled by the caller unless *PyArray_MapIterNew* is asked to allocate the result.

After preparation, getting and setting is relatively straight forward, although the different modes of iteration need to be considered. Unless there is only a single indexing array during item getting, the validity of the indices is checked beforehand. Otherwise it is handled in the inner loop itself for optimization.

## 9.1.7 Universal Functions

Universal functions are callable objects that take $N$ inputs and produce $M$ outputs by wrapping basic 1-D loops that work element-by-element into full easy-to use functions that seamlessly implement broadcasting, type-checking and buffered coercion, and output-argument handling. New universal functions are normally created in C, although there is a mechanism for creating ufuncs from Python functions (*`frompyfunc`*). The user must supply a 1-D loop that implements the basic function taking the input scalar values and placing the resulting scalars into the appropriate output slots as explained in implementation.

### Setup

Every ufunc calculation involves some overhead related to setting up the calculation. The practical significance of this overhead is that even though the actual calculation of the ufunc is very fast, you will be able to write array and type-specific code that will work faster for small arrays than the ufunc. In particular, using ufuncs to perform many calculations on 0-D arrays will be slower than other Python-based solutions (the silently-imported scalarmath module exists precisely to give array scalars the look-and-feel of ufunc based calculations with significantly reduced overhead).

When a ufunc is called, many things must be done. The information collected from these setup operations is stored in a loop-object. This loop object is a C-structure (that could become a Python object but is not initialized as such because it is only used internally). This loop object has the layout needed to be used with PyArray_Broadcast so that the broadcasting can be handled in the same way as it is handled in other sections of code.

The first thing done is to look-up in the thread-specific global dictionary the current values for the buffer-size, the error mask, and the associated error object. The state of the error mask controls what happens when an error condition is found. It should be noted that checking of the hardware error flags is only performed after each 1-D loop is executed. This means that if the input and output arrays are contiguous and of the correct type so that a single 1-D loop is performed, then the flags may not be checked until all elements of the array have been calculated. Looking up these values in a thread- specific dictionary takes time which is easily ignored for all but very small arrays.

After checking, the thread-specific global variables, the inputs are evaluated to determine how the ufunc should proceed and the input and output arrays are constructed if necessary. Any inputs which are not arrays are converted to arrays (using context if necessary). Which of the inputs are scalars (and therefore converted to 0-D arrays) is noted.

Next, an appropriate 1-D loop is selected from the 1-D loops available to the ufunc based on the input array types. This 1-D loop is selected by trying to match the signature of the data-types of the inputs against the available signatures. The signatures corresponding to built-in types are stored in the types member of the ufunc structure. The signatures corresponding to user-defined types are stored in a linked-list of function-information with the head element stored as a `CObject` in the userloops dictionary keyed by the data-type number (the first user-defined type in the argument list is used as the key). The signatures are searched until a signature is found to which the input arrays can all be cast safely (ignoring any scalar arguments which are not allowed to determine the type of the result). The implication of this search procedure is that “lesser types” should be placed below “larger types” when the signatures are stored. If no 1-D loop is found, then an error is reported. Otherwise, the argument_list is updated with the stored signature — in case casting is necessary and to fix the output types assumed by the 1-D loop.

If the ufunc has 2 inputs and 1 output and the second input is an Object array then a special-case check is performed so that NotImplemented is returned if the second input is not an ndarray, has the __array_priority__ attribute, and has an __r{op}__ special method. In this way, Python is signaled to give the other object a chance to complete the operation instead of using generic object-array calculations. This allows (for example) sparse matrices to override the multiplication operator 1-D loop.

For input arrays that are smaller than the specified buffer size, copies are made of all non-contiguous, mis-aligned, or out-of- byteorder arrays to ensure that for small arrays, a single loop is used. Then, array iterators are created for all the input arrays and the resulting collection of iterators is broadcast to a single shape.

The output arguments (if any) are then processed and any missing return arrays are constructed. If any provided output array doesn’t have the correct type (or is mis-aligned) and is smaller than the buffer size, then a new output array is constructed with the special `WRITEBACKIFCOPY` flag set. At the end of the function, *`PyArray_ResolveWritebackIfCopy`* is called so that its contents will be copied back into the output array. Iterators for the output arguments are then processed.

Finally, the decision is made about how to execute the looping mechanism to ensure that all elements of the input arrays are combined to produce the output arrays of the correct type. The options for loop execution are one-loop (for contiguous, aligned, and correct data type), strided-loop (for non-contiguous but still aligned and correct data type), and a buffered loop (for mis-aligned or incorrect data type situations). Depending on which execution method is called for, the loop is then setup and computed.

### Function call

This section describes how the basic universal function computation loop is setup and executed for each of the three different kinds of execution. If `NPY_ALLOW_THREADS` is defined during compilation, then as long as no object arrays are involved, the Python Global Interpreter Lock (GIL) is released prior to calling the loops. It is re-acquired if necessary to handle error conditions. The hardware error flags are checked only after the 1-D loop is completed.

### One Loop

This is the simplest case of all. The ufunc is executed by calling the underlying 1-D loop exactly once. This is possible only when we have aligned data of the correct type (including byte-order) for both input and output and all arrays have uniform strides (either contiguous, 0-D, or 1-D). In this case, the 1-D computational loop is called once to compute the calculation for the entire array. Note that the hardware error flags are only checked after the entire calculation is complete.

### Strided Loop

When the input and output arrays are aligned and of the correct type, but the striding is not uniform (non-contiguous and 2-D or larger), then a second looping structure is employed for the calculation. This approach converts all of the iterators for the input and output arguments to iterate over all but the largest dimension. The inner loop is then handled by the underlying 1-D computational loop. The outer loop is a standard iterator loop on the converted iterators. The hardware error flags are checked after each 1-D loop is completed.

### Buffered Loop

This is the code that handles the situation whenever the input and/or output arrays are either misaligned or of the wrong data-type (including being byte-swapped) from what the underlying 1-D loop expects. The arrays are also assumed to be non-contiguous. The code works very much like the strided-loop except for the inner 1-D loop is modified so that pre-processing is performed on the inputs and post- processing is performed on the outputs in bufsize chunks (where bufsize is a user-settable parameter). The underlying 1-D computational loop is called on data that is copied over (if it needs to be). The setup code and the loop code is considerably more complicated in this case because it has to handle:

- memory allocation of the temporary buffers
- deciding whether or not to use buffers on the input and output data (mis-aligned and/or wrong data-type)
- copying and possibly casting data for any inputs or outputs for which buffers are necessary.
- special-casing Object arrays so that reference counts are properly handled when copies and/or casts are necessary.
- breaking up the inner 1-D loop into bufsize chunks (with a possible remainder).

Again, the hardware error flags are checked at the end of each 1-D loop.

## Final output manipulation

Ufuncs allow other array-like classes to be passed seamlessly through the interface in that inputs of a particular class will induce the outputs to be of that same class. The mechanism by which this works is the following. If any of the inputs are not ndarrays and define the `__array_wrap__` method, then the class with the largest `__array_priority__` attribute determines the type of all the outputs (with the exception of any output arrays passed in). The `__array_wrap__` method of the input array will be called with the ndarray being returned from the ufunc as it's input. There are two calling styles of the `__array_wrap__` function supported. The first takes the ndarray as the first argument and a tuple of "context" as the second argument. The context is (ufunc, arguments, output argument number). This is the first call tried. If a TypeError occurs, then the function is called with just the ndarray as the first argument.

## Methods

There are three methods of ufuncs that require calculation similar to the general-purpose ufuncs. These are reduce, accumulate, and reduceat. Each of these methods requires a setup command followed by a loop. There are four loop styles possible for the methods corresponding to no-elements, one-element, strided-loop, and buffered- loop. These are the same basic loop styles as implemented for the general purpose function call except for the no-element and one- element cases which are special-cases occurring when the input array objects have 0 and 1 elements respectively.

### Setup

The setup function for all three methods is `construct_reduce`. This function creates a reducing loop object and fills it with parameters needed to complete the loop. All of the methods only work on ufuncs that take 2-inputs and return 1 output. Therefore, the underlying 1-D loop is selected assuming a signature of [ `otype`, `otype`, `otype` ] where `otype` is the requested reduction data-type. The buffer size and error handling is then retrieved from (per-thread) global storage. For small arrays that are mis-aligned or have incorrect data-type, a copy is made so that the un-buffered section of code is used. Then, the looping strategy is selected. If there is 1 element or 0 elements in the array, then a simple looping method is selected. If the array is not mis-aligned and has the correct data-type, then strided looping is selected. Otherwise, buffered looping must be performed. Looping parameters are then established, and the return array is constructed. The output array is of a different shape depending on whether the method is reduce, accumulate, or reduceat. If an output array is already provided, then it's shape is checked. If the output array is not C-contiguous, aligned, and of the correct data type, then a temporary copy is made with the WRITEBACKIFCOPY flag set. In this way, the methods will be able to work with a well-behaved output array but the result will be copied back into the true output array when `PyArray_ResolveWritebackIfCopy` is called at function completion. Finally, iterators are set up to loop over the correct axis (depending on the value of axis provided to the method) and the setup routine returns to the actual computation routine.

### Reduce

All of the ufunc methods use the same underlying 1-D computational loops with input and output arguments adjusted so that the appropriate reduction takes place. For example, the key to the functioning of reduce is that the 1-D loop is called with the output and the second input pointing to the same position in memory and both having a step- size of 0. The first input is pointing to the input array with a step- size given by the appropriate stride for the selected axis. In this way, the operation performed is

$$\begin{aligned} o &= & i[0] \\ o &= & i[k]\text{<op>}o \quad k=1\ldots N \end{aligned}$$

where $N+1$ is the number of elements in the input, $i$, $o$ is the output, and $i[k]$ is the $k^{\text{th}}$ element of $i$ along the selected axis. This basic operations is repeated for arrays with greater than 1 dimension so that the reduction takes place for every 1-D sub-array along the selected axis. An iterator with the selected dimension removed handles this looping.

For buffered loops, care must be taken to copy and cast data before the loop function is called because the underlying loop expects aligned data of the correct data-type (including byte-order). The buffered loop must handle this copying and casting prior to calling the loop function on chunks no greater than the user-specified bufsize.

### Accumulate

The accumulate function is very similar to the reduce function in that the output and the second input both point to the output. The difference is that the second input points to memory one stride behind the current output pointer. Thus, the operation performed is

$$\begin{aligned} o[0] &= & i[0] \\ o[k] &= & i[k]\text{<op>}o[k-1] \quad k=1\ldots N. \end{aligned}$$

The output has the same shape as the input and each 1-D loop operates over $N$ elements when the shape in the selected axis is $N+1$. Again, buffered loops take care to copy and cast the data before calling the underlying 1-D computational loop.

### Reduceat

The reduceat function is a generalization of both the reduce and accumulate functions. It implements a reduce over ranges of the input array specified by indices. The extra indices argument is checked to be sure that every input is not too large for the input array along the selected dimension before the loop calculations take place. The loop implementation is handled using code that is very similar to the reduce code repeated as many times as there are elements in the indices input. In particular: the first input pointer passed to the underlying 1-D computational loop points to the input array at the correct location indicated by the index array. In addition, the output pointer and the second input pointer passed to the underlying 1-D loop point to the same position in memory. The size of the 1-D computational loop is fixed to be the difference between the current index and the next index (when the current index is the last index, then the next index is assumed to be the length of the array along the selected dimension). In this way, the 1-D loop will implement a reduce over the specified indices.

Mis-aligned or a loop data-type that does not match the input and/or output data-type is handled using buffered code where-in data is copied to a temporary buffer and cast to the correct data-type if necessary prior to calling the underlying 1-D function. The temporary buffers are created in (element) sizes no bigger than the user settable buffer-size value. Thus, the loop must be flexible enough to call the underlying 1-D computational loop enough times to complete the total calculation in chunks no bigger than the buffer-size.

# 9.2 Memory Alignment

## 9.2.1 Numpy Alignment Goals

There are three use-cases related to memory alignment in numpy (as of 1.14):

1. Creating structured datatypes with fields aligned like in a C-struct.
2. Speeding up copy operations by using uint assignment in instead of memcpy
3. Guaranteeing safe aligned access for ufuncs/setitem/casting code

Numpy uses two different forms of alignment to achieve these goals: “True alignment” and “Uint alignment”.

“True” alignment refers to the architecture-dependent alignment of an equivalent C-type in C. For example, in x64 systems `numpy.float64` is equivalent to `double` in C. On most systems this has either an alignment of 4 or 8 bytes (and this can be controlled in gcc by the option `malign-double`). A variable is aligned in memory if its memory offset is a multiple of its alignment. On some systems (eg sparc) memory alignment is required, on others it gives a speedup.

“Uint” alignment depends on the size of a datatype. It is defined to be the “True alignment” of the uint used by numpy’s copy-code to copy the datatype, or undefined/unaligned if there is no equivalent uint. Currently numpy uses uint8, uint16, uint32, uint64 and uint64 to copy data of size 1,2,4,8,16 bytes respectively, and all other sized datatypes cannot be uint-aligned.

For example, on a (typical linux x64 gcc) system, the numpy `complex64` datatype is implemented as `struct { float real, imag; }`. This has “true” alignment of 4 and “uint” alignment of 8 (equal to the true alignment of `uint64`).

**Some cases where uint and true alignment are different (default gcc linux):**

> arch type true-aln uint-aln —- —- ——— ——— x86_64 complex64 4 8 x86_64 float128 16 8 x86 float96 4 -

### 9.2.2 Variables in Numpy which control and describe alignment

There are 4 relevant uses of the word `align` used in numpy:

- The `dtype.alignment` attribute (`descr->alignment` in C). This is meant to reflect the "true alignment" of the type. It has arch-dependent default values for all datatypes, with the exception of structured types created with `align=True` as described below.
- The `ALIGNED` flag of an ndarray, computed in `IsAligned` and checked by `PyArray_ISALIGNED`. This is computed from `dtype.alignment`. It is set to `True` if every item in the array is at a memory location consistent with `dtype.alignment`, which is the case if the data ptr and all strides of the array are multiples of that alignment.
- The `align` keyword of the dtype constructor, which only affects structured arrays. If the structure's field offsets are not manually provided numpy determines offsets automatically. In that case, `align=True` pads the structure so that each field is "true" aligned in memory and sets `dtype.alignment` to be the largest of the field "true" alignments. This is like what C-structs usually do. Otherwise if offsets or itemsize were manually provided `align=True` simply checks that all the fields are "true" aligned and that the total itemsize is a multiple of the largest field alignment. In either case `dtype.isalignedstruct` is also set to True.
- `IsUintAligned` is used to determine if an ndarray is "uint aligned" in an analogous way to how `IsAligned` checks for true-alignment.

### 9.2.3 Consequences of alignment

Here is how the variables above are used:

1. Creating aligned structs: In order to know how to offset a field when `align=True`, numpy looks up `field.dtype.alignment`. This includes fields which are nested structured arrays.
2. Ufuncs: If the `ALIGNED` flag of an array is False, ufuncs will buffer/cast the array before evaluation. This is needed since ufunc inner loops access raw elements directly, which might fail on some archs if the elements are not true-aligned.
3. Getitem/setitem/copyswap function: Similar to ufuncs, these functions generally have two code paths. If `ALIGNED` is False they will use a code path that buffers the arguments so they are true-aligned.
4. Strided copy code: Here, "uint alignment" is used instead. If the itemsize of an array is equal to 1, 2, 4, 8 or 16 bytes and the array is uint aligned then instead numpy will do `*(uintN*)dst) = *(uintN*)src)` for appropriate N. Otherwise numpy copies by doing `memcpy(dst, src, N)`.
5. Nditer code: Since this often calls the strided copy code, it must check for "uint alignment".
6. Cast code: This checks for "true" alignment, as it does `*dst = CASTFUNC(*src)` if aligned. Otherwise, it does `memmove(srcval, src); dstval = CASTFUNC(srcval); memmove(dst, dstval)` where dstval/srcval are aligned.

Note that the strided-copy and strided-cast code are deeply intertwined and so any arrays being processed by them must be both uint and true aligned, even though the copy-code only needs uint alignment and the cast code only true alignment. If there is ever a big rewrite of this code it would be good to allow them to use different alignments.

## 9.3 Internal organization of numpy arrays

It helps to understand a bit about how numpy arrays are handled under the covers to help understand numpy better. This section will not go into great detail. Those wishing to understand the full details are referred to Travis Oliphant's book "Guide to NumPy".

NumPy arrays consist of two major components, the raw array data (from now on, referred to as the data buffer), and the information about the raw array data. The data buffer is typically what people think of as arrays in C or Fortran, a contiguous (and fixed) block of memory containing fixed sized data items. NumPy also contains a significant set of data that describes how to interpret the data in the data buffer. This extra information contains (among other things):

1) The basic data element's size in bytes
2) The start of the data within the data buffer (an offset relative to the beginning of the data buffer).
3) The number of dimensions and the size of each dimension
4) The separation between elements for each dimension (the 'stride'). This does not have to be a multiple of the element size
5) The byte order of the data (which may not be the native byte order)
6) Whether the buffer is read-only
7) Information (via the dtype object) about the interpretation of the basic data element. The basic data element may be as simple as a int or a float, or it may be a compound object (e.g., struct-like), a fixed character field, or Python object pointers.
8) Whether the array is to interpreted as C-order or Fortran-order.

This arrangement allow for very flexible use of arrays. One thing that it allows is simple changes of the metadata to change the interpretation of the array buffer. Changing the byteorder of the array is a simple change involving no rearrangement of the data. The shape of the array can be changed very easily without changing anything in the data buffer or any data copying at all

Among other things that are made possible is one can create a new array metadata object that uses the same data buffer to create a new view of that data buffer that has a different interpretation of the buffer (e.g., different shape, offset, byte order, strides, etc) but shares the same data bytes. Many operations in numpy do just this such as slices. Other operations, such as transpose, don't move data elements around in the array, but rather change the information about the shape and strides so that the indexing of the array changes, but the data in the doesn't move.

Typically these new versions of the array metadata but the same data buffer are new 'views' into the data buffer. There is a different ndarray object, but it uses the same data buffer. This is why it is necessary to force copies through use of the .copy() method if one really wants to make a new and independent copy of the data buffer.

New views into arrays mean the object reference counts for the data buffer increase. Simply doing away with the original array object will not remove the data buffer if other views of it still exist.

## 9.4 Multidimensional Array Indexing Order Issues

What is the right way to index multi-dimensional arrays? Before you jump to conclusions about the one and true way to index multi-dimensional arrays, it pays to understand why this is a confusing issue. This section will try to explain in detail how numpy indexing works and why we adopt the convention we do for images, and when it may be appropriate to adopt other conventions.

The first thing to understand is that there are two conflicting conventions for indexing 2-dimensional arrays. Matrix notation uses the first index to indicate which row is being selected and the second index to indicate which column is selected. This is opposite the geometrically oriented-convention for images where people generally think the first index represents x position (i.e., column) and the second represents y position (i.e., row). This alone is the source of much confusion; matrix-oriented users and image-oriented users expect two different things with regard to indexing.

The second issue to understand is how indices correspond to the order the array is stored in memory. In Fortran the first index is the most rapidly varying index when moving through the elements of a two dimensional array as it is stored in memory. If you adopt the matrix convention for indexing, then this means the matrix is stored one column at a time (since the first index moves to the next row as it changes). Thus Fortran is considered a Column-major language. C has just the opposite convention. In C, the last index changes most rapidly as one moves through the array as stored in memory. Thus C is a Row-major language. The matrix is stored by rows. Note that in both cases it presumes that the matrix convention for indexing is being used, i.e., for both Fortran and C, the first index is the row. Note this convention implies that the indexing convention is invariant and that the data order changes to keep that so.

But that's not the only way to look at it. Suppose one has large two-dimensional arrays (images or matrices) stored in data files. Suppose the data are stored by rows rather than by columns. If we are to preserve our index convention (whether matrix or image) that means that depending on the language we use, we may be forced to reorder the data if it is read into memory to preserve our indexing convention. For example if we read row-ordered data into memory without reordering, it will match the matrix indexing convention for C, but not for Fortran. Conversely, it will match the image indexing convention for Fortran, but not for C. For C, if one is using data stored in row order, and one wants to preserve the image index convention, the data must be reordered when reading into memory.

In the end, which you do for Fortran or C depends on which is more important, not reordering data or preserving the indexing convention. For large images, reordering data is potentially expensive, and often the indexing convention is inverted to avoid that.

The situation with numpy makes this issue yet more complicated. The internal machinery of numpy arrays is flexible enough to accept any ordering of indices. One can simply reorder indices by manipulating the internal stride information for arrays without reordering the data at all. NumPy will know how to map the new index order to the data without moving the data.

So if this is true, why not choose the index order that matches what you most expect? In particular, why not define row-ordered images to use the image convention? (This is sometimes referred to as the Fortran convention vs the C convention, thus the 'C' and 'FORTRAN' order options for array ordering in numpy.) The drawback of doing this is potential performance penalties. It's common to access the data sequentially, either implicitly in array operations or explicitly by looping over rows of an image. When that is done, then the data will be accessed in non-optimal order. As the first index is incremented, what is actually happening is that elements spaced far apart in memory are being sequentially accessed, with usually poor memory access speeds. For example, for a two dimensional image 'im' defined so that im[0, 10] represents the value at x=0, y=10. To be consistent with usual Python behavior then im[0] would represent a column at x=0. Yet that data would be spread over the whole array since the data are stored in row order. Despite the flexibility of numpy's indexing, it can't really paper over the fact basic operations are rendered inefficient because of data order or that getting contiguous subarrays is still awkward (e.g., im[:,0] for the first row, vs im[0]), thus one can't use an idiom such as for row in im; for col in im does work, but doesn't yield contiguous column data.

As it turns out, numpy is smart enough when dealing with ufuncs to determine which index is the most rapidly varying one in memory and uses that for the innermost loop. Thus for ufuncs there is no large intrinsic advantage to either approach in most cases. On the other hand, use of .flat with an FORTRAN ordered array will lead to non-optimal memory access as adjacent elements in the flattened array (iterator, actually) are not contiguous in memory.

Indeed, the fact is that Python indexing on lists and other sequences naturally leads to an outside-to inside ordering (the first index gets the largest grouping, the next the next largest, and the last gets the smallest element). Since image data are normally stored by rows, this corresponds to position within rows being the last item indexed.

If you do want to use Fortran ordering realize that there are two approaches to consider: 1) accept that the first index is just not the most rapidly changing in memory and have all your I/O routines reorder your data when going from memory to disk or visa versa, or use numpy's mechanism for mapping the first index to the most rapidly varying data. We recommend the former if possible. The disadvantage of the latter is that many of numpy's functions will yield arrays without Fortran ordering unless you are careful to use the 'order' keyword. Doing this would be highly inconvenient.

Otherwise we recommend simply learning to reverse the usual order of indices when accessing elements of an array.

Granted, it goes against the grain, but it is more in line with Python semantics and the natural order of the data.

# CHAPTER

# TEN

# NUMPY AND SWIG

### 10.0.1 Introduction

The Simple Wrapper and Interface Generator (or SWIG) is a powerful tool for generating wrapper code for interfacing to a wide variety of scripting languages. SWIG can parse header files, and using only the code prototypes, create an interface to the target language. But SWIG is not omnipotent. For example, it cannot know from the prototype:

```
double rms(double* seq, int n);
```

what exactly `seq` is. Is it a single value to be altered in-place? Is it an array, and if so what is its length? Is it input-only? Output-only? Input-output? SWIG cannot determine these details, and does not attempt to do so.

If we designed `rms`, we probably made it a routine that takes an input-only array of length `n` of `double` values called `seq` and returns the root mean square. The default behavior of SWIG, however, will be to create a wrapper function that compiles, but is nearly impossible to use from the scripting language in the way the C routine was intended.

For Python, the preferred way of handling contiguous (or technically, *strided*) blocks of homogeneous data is with NumPy, which provides full object-oriented access to multidimensional arrays of data. Therefore, the most logical Python interface for the `rms` function would be (including doc string):

```
def rms(seq):
    """
    rms: return the root mean square of a sequence
    rms(numpy.ndarray) -> double
    rms(list) -> double
    rms(tuple) -> double
    """
```

where `seq` would be a NumPy array of `double` values, and its length `n` would be extracted from `seq` internally before being passed to the C routine. Even better, since NumPy supports construction of arrays from arbitrary Python sequences, `seq` itself could be a nearly arbitrary sequence (so long as each element can be converted to a `double`) and the wrapper code would internally convert it to a NumPy array before extracting its data and length.

SWIG allows these types of conversions to be defined via a mechanism called *typemaps*. This document provides information on how to use `numpy.i`, a SWIG interface file that defines a series of typemaps intended to make the type of array-related conversions described above relatively simple to implement. For example, suppose that the `rms` function prototype defined above was in a header file named `rms.h`. To obtain the Python interface discussed above, your SWIG interface file would need the following:

```
%{
#define SWIG_FILE_WITH_INIT
#include "rms.h"
%}

```

(continues on next page)

(continued from previous page)

```
%include "numpy.i"

%init %{
import_array();
%}

%apply (double* IN_ARRAY1, int DIM1) {(double* seq, int n)};
%include "rms.h"
```

Typemaps are keyed off a list of one or more function arguments, either by type or by type and name. We will refer to such lists as *signatures*. One of the many typemaps defined by `numpy.i` is used above and has the signature `(double* IN_ARRAY1, int DIM1)`. The argument names are intended to suggest that the `double*` argument is an input array of one dimension and that the `int` represents the size of that dimension. This is precisely the pattern in the `rms` prototype.

Most likely, no actual prototypes to be wrapped will have the argument names `IN_ARRAY1` and `DIM1`. We use the SWIG `%apply` directive to apply the typemap for one-dimensional input arrays of type `double` to the actual prototype used by `rms`. Using `numpy.i` effectively, therefore, requires knowing what typemaps are available and what they do.

A SWIG interface file that includes the SWIG directives given above will produce wrapper code that looks something like:

```
 1 PyObject *_wrap_rms(PyObject *args) {
 2   PyObject *resultobj = 0;
 3   double *arg1 = (double *) 0 ;
 4   int arg2 ;
 5   double result;
 6   PyArrayObject *array1 = NULL ;
 7   int is_new_object1 = 0 ;
 8   PyObject * obj0 = 0 ;
 9
10   if (!PyArg_ParseTuple(args,(char *)"O:rms",&obj0)) SWIG_fail;
11   {
12     array1 = obj_to_array_contiguous_allow_conversion(
13                  obj0, NPY_DOUBLE, &is_new_object1);
14     npy_intp size[1] = {
15       -1
16     };
17     if (!array1 || !require_dimensions(array1, 1) ||
18         !require_size(array1, size, 1)) SWIG_fail;
19     arg1 = (double*) array1->data;
20     arg2 = (int) array1->dimensions[0];
21   }
22   result = (double)rms(arg1,arg2);
23   resultobj = SWIG_From_double((double)(result));
24   {
25     if (is_new_object1 && array1) Py_DECREF(array1);
26   }
27   return resultobj;
28 fail:
29   {
30     if (is_new_object1 && array1) Py_DECREF(array1);
31   }
32   return NULL;
33 }
```

The typemaps from `numpy.i` are responsible for the following lines of code: 12–20, 25 and 30. Line 10 parses the input to the `rms` function. From the format string `"O:rms"`, we can see that the argument list is expected to be a single Python object (specified by the `O` before the colon) and whose pointer is stored in `obj0`. A number of functions, supplied by `numpy.i`, are called to make and check the (possible) conversion from a generic Python object to a NumPy array. These functions are explained in the section *Helper Functions*, but hopefully their names are self-explanatory. At line 12 we use `obj0` to construct a NumPy array. At line 17, we check the validity of the result: that it is non-null and that it has a single dimension of arbitrary length. Once these states are verified, we extract the data buffer and length in lines 19 and 20 so that we can call the underlying C function at line 22. Line 25 performs memory management for the case where we have created a new array that is no longer needed.

This code has a significant amount of error handling. Note the `SWIG_fail` is a macro for `goto fail`, referring to the label at line 28. If the user provides the wrong number of arguments, this will be caught at line 10. If construction of the NumPy array fails or produces an array with the wrong number of dimensions, these errors are caught at line 17. And finally, if an error is detected, memory is still managed correctly at line 30.

Note that if the C function signature was in a different order:

```
double rms(int n, double* seq);
```

that SWIG would not match the typemap signature given above with the argument list for `rms`. Fortunately, `numpy.i` has a set of typemaps with the data pointer given last:

```
%apply (int DIM1, double* IN_ARRAY1) {(int n, double* seq)};
```

This simply has the effect of switching the definitions of `arg1` and `arg2` in lines 3 and 4 of the generated code above, and their assignments in lines 19 and 20.

### 10.0.2 Using numpy.i

The `numpy.i` file is currently located in the `tools/swig` sub-directory under the `numpy` installation directory. Typically, you will want to copy it to the directory where you are developing your wrappers.

A simple module that only uses a single SWIG interface file should include the following:

```
%{
#define SWIG_FILE_WITH_INIT
%}
%include "numpy.i"
%init %{
import_array();
%}
```

Within a compiled Python module, `import_array()` should only get called once. This could be in a C/C++ file that you have written and is linked to the module. If this is the case, then none of your interface files should `#define SWIG_FILE_WITH_INIT` or call `import_array()`. Or, this initialization call could be in a wrapper file generated by SWIG from an interface file that has the `%init` block as above. If this is the case, and you have more than one SWIG interface file, then only one interface file should `#define SWIG_FILE_WITH_INIT` and call `import_array()`.

### 10.0.3 Available Typemaps

The typemap directives provided by `numpy.i` for arrays of different data types, say `double` and `int`, and dimensions of different types, say `int` or `long`, are identical to one another except for the C and NumPy type specifications. The typemaps are therefore implemented (typically behind the scenes) via a macro:

```
%numpy_typemaps(DATA_TYPE, DATA_TYPECODE, DIM_TYPE)
```

that can be invoked for appropriate `(DATA_TYPE, DATA_TYPECODE, DIM_TYPE)` triplets. For example:

```
%numpy_typemaps(double, NPY_DOUBLE, int)
%numpy_typemaps(int,    NPY_INT   , int)
```

The `numpy.i` interface file uses the `%numpy_typemaps` macro to implement typemaps for the following C data types and `int` dimension types:

- `signed char`
- `unsigned char`
- `short`
- `unsigned short`
- `int`
- `unsigned int`
- `long`
- `unsigned long`
- `long long`
- `unsigned long long`
- `float`
- `double`

In the following descriptions, we reference a generic `DATA_TYPE`, which could be any of the C data types listed above, and `DIM_TYPE` which should be one of the many types of integers.

The typemap signatures are largely differentiated on the name given to the buffer pointer. Names with `FARRAY` are for Fortran-ordered arrays, and names with `ARRAY` are for C-ordered (or 1D arrays).

#### Input Arrays

Input arrays are defined as arrays of data that are passed into a routine but are not altered in-place or returned to the user. The Python input array is therefore allowed to be almost any Python sequence (such as a list) that can be converted to the requested type of array. The input array signatures are

1D:

- `( DATA_TYPE IN_ARRAY1[ANY] )`
- `( DATA_TYPE* IN_ARRAY1, int DIM1 )`
- `( int DIM1, DATA_TYPE* IN_ARRAY1 )`

2D:

- `( DATA_TYPE IN_ARRAY2[ANY][ANY] )`

- `( DATA_TYPE* IN_ARRAY2, int DIM1, int DIM2 )`
- `( int DIM1, int DIM2, DATA_TYPE* IN_ARRAY2 )`
- `( DATA_TYPE* IN_FARRAY2, int DIM1, int DIM2 )`
- `( int DIM1, int DIM2, DATA_TYPE* IN_FARRAY2 )`

3D:

- `( DATA_TYPE IN_ARRAY3[ANY][ANY][ANY] )`
- `( DATA_TYPE* IN_ARRAY3, int DIM1, int DIM2, int DIM3 )`
- `( int DIM1, int DIM2, int DIM3, DATA_TYPE* IN_ARRAY3 )`
- `( DATA_TYPE* IN_FARRAY3, int DIM1, int DIM2, int DIM3 )`
- `( int DIM1, int DIM2, int DIM3, DATA_TYPE* IN_FARRAY3 )`

4D:

- `(DATA_TYPE IN_ARRAY4[ANY][ANY][ANY][ANY])`
- `(DATA_TYPE* IN_ARRAY4, DIM_TYPE DIM1, DIM_TYPE DIM2, DIM_TYPE DIM3, DIM_TYPE DIM4)`
- `(DIM_TYPE DIM1, DIM_TYPE DIM2, DIM_TYPE DIM3, , DIM_TYPE DIM4, DATA_TYPE* IN_ARRAY4)`
- `(DATA_TYPE* IN_FARRAY4, DIM_TYPE DIM1, DIM_TYPE DIM2, DIM_TYPE DIM3, DIM_TYPE DIM4)`
- `(DIM_TYPE DIM1, DIM_TYPE DIM2, DIM_TYPE DIM3, DIM_TYPE DIM4, DATA_TYPE* IN_FARRAY4)`

The first signature listed, `( DATA_TYPE IN_ARRAY[ANY] )` is for one-dimensional arrays with hard-coded dimensions. Likewise, `( DATA_TYPE IN_ARRAY2[ANY][ANY] )` is for two-dimensional arrays with hard-coded dimensions, and similarly for three-dimensional.

### In-Place Arrays

In-place arrays are defined as arrays that are modified in-place. The input values may or may not be used, but the values at the time the function returns are significant. The provided Python argument must therefore be a NumPy array of the required type. The in-place signatures are

1D:

- `( DATA_TYPE INPLACE_ARRAY1[ANY] )`
- `( DATA_TYPE* INPLACE_ARRAY1, int DIM1 )`
- `( int DIM1, DATA_TYPE* INPLACE_ARRAY1 )`

2D:

- `( DATA_TYPE INPLACE_ARRAY2[ANY][ANY] )`
- `( DATA_TYPE* INPLACE_ARRAY2, int DIM1, int DIM2 )`
- `( int DIM1, int DIM2, DATA_TYPE* INPLACE_ARRAY2 )`
- `( DATA_TYPE* INPLACE_FARRAY2, int DIM1, int DIM2 )`
- `( int DIM1, int DIM2, DATA_TYPE* INPLACE_FARRAY2 )`

3D:

- `( DATA_TYPE INPLACE_ARRAY3[ANY][ANY][ANY] )`
- `( DATA_TYPE* INPLACE_ARRAY3, int DIM1, int DIM2, int DIM3 )`
- `( int DIM1, int DIM2, int DIM3, DATA_TYPE* INPLACE_ARRAY3 )`
- `( DATA_TYPE* INPLACE_FARRAY3, int DIM1, int DIM2, int DIM3 )`
- `( int DIM1, int DIM2, int DIM3, DATA_TYPE* INPLACE_FARRAY3 )`

4D:

- `(DATA_TYPE INPLACE_ARRAY4[ANY][ANY][ANY][ANY])`
- `(DATA_TYPE* INPLACE_ARRAY4, DIM_TYPE DIM1, DIM_TYPE DIM2, DIM_TYPE DIM3, DIM_TYPE DIM4)`
- `(DIM_TYPE DIM1, DIM_TYPE DIM2, DIM_TYPE DIM3, , DIM_TYPE DIM4, DATA_TYPE* INPLACE_ARRAY4)`
- `(DATA_TYPE* INPLACE_FARRAY4, DIM_TYPE DIM1, DIM_TYPE DIM2, DIM_TYPE DIM3, DIM_TYPE DIM4)`
- `(DIM_TYPE DIM1, DIM_TYPE DIM2, DIM_TYPE DIM3, DIM_TYPE DIM4, DATA_TYPE* INPLACE_FARRAY4)`

These typemaps now check to make sure that the `INPLACE_ARRAY` arguments use native byte ordering. If not, an exception is raised.

There is also a “flat” in-place array for situations in which you would like to modify or process each element, regardless of the number of dimensions. One example is a “quantization” function that quantizes each element of an array in-place, be it 1D, 2D or whatever. This form checks for continuity but allows either C or Fortran ordering.

ND:

- `(DATA_TYPE* INPLACE_ARRAY_FLAT, DIM_TYPE DIM_FLAT)`

### Argout Arrays

Argout arrays are arrays that appear in the input arguments in C, but are in fact output arrays. This pattern occurs often when there is more than one output variable and the single return argument is therefore not sufficient. In Python, the conventional way to return multiple arguments is to pack them into a sequence (tuple, list, etc.) and return the sequence. This is what the argout typemaps do. If a wrapped function that uses these argout typemaps has more than one return argument, they are packed into a tuple or list, depending on the version of Python. The Python user does not pass these arrays in, they simply get returned. For the case where a dimension is specified, the python user must provide that dimension as an argument. The argout signatures are

1D:

- `( DATA_TYPE ARGOUT_ARRAY1[ANY] )`
- `( DATA_TYPE* ARGOUT_ARRAY1, int DIM1 )`
- `( int DIM1, DATA_TYPE* ARGOUT_ARRAY1 )`

2D:

- `( DATA_TYPE ARGOUT_ARRAY2[ANY][ANY] )`

3D:

- `( DATA_TYPE ARGOUT_ARRAY3[ANY][ANY][ANY] )`

4D:

- `( DATA_TYPE ARGOUT_ARRAY4[ANY][ANY][ANY][ANY] )`

These are typically used in situations where in C/C++, you would allocate a(n) array(s) on the heap, and call the function to fill the array(s) values. In Python, the arrays are allocated for you and returned as new array objects.

Note that we support `DATA_TYPE*` argout typemaps in 1D, but not 2D or 3D. This is because of a quirk with the SWIG typemap syntax and cannot be avoided. Note that for these types of 1D typemaps, the Python function will take a single argument representing `DIM1`.

### Argout View Arrays

Argoutview arrays are for when your C code provides you with a view of its internal data and does not require any memory to be allocated by the user. This can be dangerous. There is almost no way to guarantee that the internal data from the C code will remain in existence for the entire lifetime of the NumPy array that encapsulates it. If the user destroys the object that provides the view of the data before destroying the NumPy array, then using that array may result in bad memory references or segmentation faults. Nevertheless, there are situations, working with large data sets, where you simply have no other choice.

The C code to be wrapped for argoutview arrays are characterized by pointers: pointers to the dimensions and double pointers to the data, so that these values can be passed back to the user. The argoutview typemap signatures are therefore

1D:

- `( DATA_TYPE** ARGOUTVIEW_ARRAY1, DIM_TYPE* DIM1 )`
- `( DIM_TYPE* DIM1, DATA_TYPE** ARGOUTVIEW_ARRAY1 )`

2D:

- `( DATA_TYPE** ARGOUTVIEW_ARRAY2, DIM_TYPE* DIM1, DIM_TYPE* DIM2 )`
- `( DIM_TYPE* DIM1, DIM_TYPE* DIM2, DATA_TYPE** ARGOUTVIEW_ARRAY2 )`
- `( DATA_TYPE** ARGOUTVIEW_FARRAY2, DIM_TYPE* DIM1, DIM_TYPE* DIM2 )`
- `( DIM_TYPE* DIM1, DIM_TYPE* DIM2, DATA_TYPE** ARGOUTVIEW_FARRAY2 )`

3D:

- `( DATA_TYPE** ARGOUTVIEW_ARRAY3, DIM_TYPE* DIM1, DIM_TYPE* DIM2, DIM_TYPE* DIM3)`
- `( DIM_TYPE* DIM1, DIM_TYPE* DIM2, DIM_TYPE* DIM3, DATA_TYPE** ARGOUTVIEW_ARRAY3)`
- `( DATA_TYPE** ARGOUTVIEW_FARRAY3, DIM_TYPE* DIM1, DIM_TYPE* DIM2, DIM_TYPE* DIM3)`
- `( DIM_TYPE* DIM1, DIM_TYPE* DIM2, DIM_TYPE* DIM3, DATA_TYPE** ARGOUTVIEW_FARRAY3)`

4D:

- `(DATA_TYPE** ARGOUTVIEW_ARRAY4, DIM_TYPE* DIM1, DIM_TYPE* DIM2, DIM_TYPE* DIM3, DIM_TYPE* DIM4)`
- `(DIM_TYPE* DIM1, DIM_TYPE* DIM2, DIM_TYPE* DIM3, DIM_TYPE* DIM4, DATA_TYPE** ARGOUTVIEW_ARRAY4)`
- `(DATA_TYPE** ARGOUTVIEW_FARRAY4, DIM_TYPE* DIM1, DIM_TYPE* DIM2, DIM_TYPE* DIM3, DIM_TYPE* DIM4)`
- `(DIM_TYPE* DIM1, DIM_TYPE* DIM2, DIM_TYPE* DIM3, DIM_TYPE* DIM4, DATA_TYPE** ARGOUTVIEW_FARRAY4)`

Note that arrays with hard-coded dimensions are not supported. These cannot follow the double pointer signatures of these typemaps.

### Memory Managed Argout View Arrays

A recent addition to `numpy.i` are typemaps that permit argout arrays with views into memory that is managed. See the discussion here.

1D:

- `(DATA_TYPE** ARGOUTVIEWM_ARRAY1, DIM_TYPE* DIM1)`
- `(DIM_TYPE* DIM1, DATA_TYPE** ARGOUTVIEWM_ARRAY1)`

2D:

- `(DATA_TYPE** ARGOUTVIEWM_ARRAY2, DIM_TYPE* DIM1, DIM_TYPE* DIM2)`
- `(DIM_TYPE* DIM1, DIM_TYPE* DIM2, DATA_TYPE** ARGOUTVIEWM_ARRAY2)`
- `(DATA_TYPE** ARGOUTVIEWM_FARRAY2, DIM_TYPE* DIM1, DIM_TYPE* DIM2)`
- `(DIM_TYPE* DIM1, DIM_TYPE* DIM2, DATA_TYPE** ARGOUTVIEWM_FARRAY2)`

3D:

- `(DATA_TYPE** ARGOUTVIEWM_ARRAY3, DIM_TYPE* DIM1, DIM_TYPE* DIM2, DIM_TYPE* DIM3)`
- `(DIM_TYPE* DIM1, DIM_TYPE* DIM2, DIM_TYPE* DIM3, DATA_TYPE** ARGOUTVIEWM_ARRAY3)`
- `(DATA_TYPE** ARGOUTVIEWM_FARRAY3, DIM_TYPE* DIM1, DIM_TYPE* DIM2, DIM_TYPE* DIM3)`
- `(DIM_TYPE* DIM1, DIM_TYPE* DIM2, DIM_TYPE* DIM3, DATA_TYPE** ARGOUTVIEWM_FARRAY3)`

4D:

- `(DATA_TYPE** ARGOUTVIEWM_ARRAY4, DIM_TYPE* DIM1, DIM_TYPE* DIM2, DIM_TYPE* DIM3, DIM_TYPE* DIM4)`
- `(DIM_TYPE* DIM1, DIM_TYPE* DIM2, DIM_TYPE* DIM3, DIM_TYPE* DIM4, DATA_TYPE** ARGOUTVIEWM_ARRAY4)`
- `(DATA_TYPE** ARGOUTVIEWM_FARRAY4, DIM_TYPE* DIM1, DIM_TYPE* DIM2, DIM_TYPE* DIM3, DIM_TYPE* DIM4)`
- `(DIM_TYPE* DIM1, DIM_TYPE* DIM2, DIM_TYPE* DIM3, DIM_TYPE* DIM4, DATA_TYPE** ARGOUTVIEWM_FARRAY4)`

### Output Arrays

The `numpy.i` interface file does not support typemaps for output arrays, for several reasons. First, C/C++ return arguments are limited to a single value. This prevents obtaining dimension information in a general way. Second, arrays with hard-coded lengths are not permitted as return arguments. In other words:

```
double[3] newVector(double x, double y, double z);
```

is not legal C/C++ syntax. Therefore, we cannot provide typemaps of the form:

```
%typemap(out) (TYPE[ANY]);
```

If you run into a situation where a function or method is returning a pointer to an array, your best bet is to write your own version of the function to be wrapped, either with `%extend` for the case of class methods or `%ignore` and `%rename` for the case of functions.

### Other Common Types: bool

Note that C++ type `bool` is not supported in the list in the *Available Typemaps* section. NumPy bools are a single byte, while the C++ `bool` is four bytes (at least on my system). Therefore:

```
%numpy_typemaps(bool, NPY_BOOL, int)
```

will result in typemaps that will produce code that reference improper data lengths. You can implement the following macro expansion:

```
%numpy_typemaps(bool, NPY_UINT, int)
```

to fix the data length problem, and *Input Arrays* will work fine, but *In-Place Arrays* might fail type-checking.

### Other Common Types: complex

Typemap conversions for complex floating-point types is also not supported automatically. This is because Python and NumPy are written in C, which does not have native complex types. Both Python and NumPy implement their own (essentially equivalent) `struct` definitions for complex variables:

```
/* Python */
typedef struct {double real; double imag;} Py_complex;

/* NumPy */
typedef struct {float  real, imag;} npy_cfloat;
typedef struct {double real, imag;} npy_cdouble;
```

We could have implemented:

```
%numpy_typemaps(Py_complex , NPY_CDOUBLE, int)
%numpy_typemaps(npy_cfloat , NPY_CFLOAT , int)
%numpy_typemaps(npy_cdouble, NPY_CDOUBLE, int)
```

which would have provided automatic type conversions for arrays of type `Py_complex`, `npy_cfloat` and `npy_cdouble`. However, it seemed unlikely that there would be any independent (non-Python, non-NumPy) application code that people would be using SWIG to generate a Python interface to, that also used these definitions for complex types. More likely, these application codes will define their own complex types, or in the case of C++, use `std::complex`. Assuming these data structures are compatible with Python and NumPy complex types, `%numpy_typemap` expansions as above (with the user's complex type substituted for the first argument) should work.

## 10.0.4 NumPy Array Scalars and SWIG

SWIG has sophisticated type checking for numerical types. For example, if your C/C++ routine expects an integer as input, the code generated by SWIG will check for both Python integers and Python long integers, and raise an overflow error if the provided Python integer is too big to cast down to a C integer. With the introduction of NumPy scalar arrays into your Python code, you might conceivably extract an integer from a NumPy array and attempt to pass this to a SWIG-wrapped C/C++ function that expects an `int`, but the SWIG type checking will not recognize the NumPy array scalar as an integer. (Often, this does in fact work – it depends on whether NumPy recognizes the integer type you are using as inheriting from the Python integer type on the platform you are using. Sometimes, this means that code that works on a 32-bit machine will fail on a 64-bit machine.)

If you get a Python error that looks like the following:

```
TypeError: in method 'MyClass_MyMethod', argument 2 of type 'int'
```

and the argument you are passing is an integer extracted from a NumPy array, then you have stumbled upon this problem. The solution is to modify the SWIG type conversion system to accept NumPy array scalars in addition to the standard integer types. Fortunately, this capability has been provided for you. Simply copy the file:

```
pyfragments.swg
```

to the working build directory for you project, and this problem will be fixed. It is suggested that you do this anyway, as it only increases the capabilities of your Python interface.

### Why is There a Second File?

The SWIG type checking and conversion system is a complicated combination of C macros, SWIG macros, SWIG typemaps and SWIG fragments. Fragments are a way to conditionally insert code into your wrapper file if it is needed, and not insert it if not needed. If multiple typemaps require the same fragment, the fragment only gets inserted into your wrapper code once.

There is a fragment for converting a Python integer to a C `long`. There is a different fragment that converts a Python integer to a C `int`, that calls the routine defined in the `long` fragment. We can make the changes we want here by changing the definition for the `long` fragment. SWIG determines the active definition for a fragment using a "first come, first served" system. That is, we need to define the fragment for `long` conversions prior to SWIG doing it internally. SWIG allows us to do this by putting our fragment definitions in the file `pyfragments.swg`. If we were to put the new fragment definitions in `numpy.i`, they would be ignored.

## 10.0.5 Helper Functions

The `numpy.i` file contains several macros and routines that it uses internally to build its typemaps. However, these functions may be useful elsewhere in your interface file. These macros and routines are implemented as fragments, which are described briefly in the previous section. If you try to use one or more of the following macros or functions, but your compiler complains that it does not recognize the symbol, then you need to force these fragments to appear in your code using:

```
%fragment("NumPy_Fragments");
```

in your SWIG interface file.

## Macros

**is_array(a)**

Evaluates as true if `a` is non-`NULL` and can be cast to a `PyArrayObject*`.

**array_type(a)**

Evaluates to the integer data type code of `a`, assuming `a` can be cast to a `PyArrayObject*`.

**array_numdims(a)**

Evaluates to the integer number of dimensions of `a`, assuming `a` can be cast to a `PyArrayObject*`.

**array_dimensions(a)**

Evaluates to an array of type `npy_intp` and length `array_numdims(a)`, giving the lengths of all of the dimensions of `a`, assuming `a` can be cast to a `PyArrayObject*`.

**array_size(a,i)**

Evaluates to the `i`-th dimension size of `a`, assuming `a` can be cast to a `PyArrayObject*`.

**array_strides(a)**

Evaluates to an array of type `npy_intp` and length `array_numdims(a)`, giving the stridess of all of the dimensions of `a`, assuming `a` can be cast to a `PyArrayObject*`. A stride is the distance in bytes between an element and its immediate neighbor along the same axis.

**array_stride(a,i)**

Evaluates to the `i`-th stride of `a`, assuming `a` can be cast to a `PyArrayObject*`.

**array_data(a)**

Evaluates to a pointer of type `void*` that points to the data buffer of `a`, assuming `a` can be cast to a `PyArrayObject*`.

**array_descr(a)**

Returns a borrowed reference to the dtype property (`PyArray_Descr*`) of `a`, assuming `a` can be cast to a `PyArrayObject*`.

**array_flags(a)**

Returns an integer representing the flags of `a`, assuming `a` can be cast to a `PyArrayObject*`.

**array_enableflags(a,f)**

Sets the flag represented by `f` of `a`, assuming `a` can be cast to a `PyArrayObject*`.

**array_is_contiguous(a)**

Evaluates as true if `a` is a contiguous array. Equivalent to `(PyArray_ISCONTIGUOUS(a))`.

**array_is_native(a)**

Evaluates as true if the data buffer of `a` uses native byte order. Equivalent to `(PyArray_ISNOTSWAPPED(a))`.

**array_is_fortran(a)**

Evaluates as true if `a` is FORTRAN ordered.

### Routines

**pytype_string()**

Return type: `const char*`

Arguments:

- `PyObject* py_obj`, a general Python object.

Return a string describing the type of `py_obj`.

**typecode_string()**

Return type: `const char*`

Arguments:

- `int typecode`, a NumPy integer typecode.

Return a string describing the type corresponding to the NumPy `typecode`.

**type_match()**

Return type: `int`

Arguments:

- `int actual_type`, the NumPy typecode of a NumPy array.
- `int desired_type`, the desired NumPy typecode.

Make sure that `actual_type` is compatible with `desired_type`. For example, this allows character and byte types, or int and long types, to match. This is now equivalent to `PyArray_EquivTypenums()`.

**obj_to_array_no_conversion()**

Return type: `PyArrayObject*`

Arguments:

- `PyObject* input`, a general Python object.
- `int typecode`, the desired NumPy typecode.

Cast `input` to a `PyArrayObject*` if legal, and ensure that it is of type `typecode`. If `input` cannot be cast, or the `typecode` is wrong, set a Python error and return `NULL`.

**obj_to_array_allow_conversion()**

Return type: `PyArrayObject*`

Arguments:

- `PyObject* input`, a general Python object.
- `int typecode`, the desired NumPy typecode of the resulting array.
- `int* is_new_object`, returns a value of 0 if no conversion performed, else 1.

Convert `input` to a NumPy array with the given `typecode`. On success, return a valid `PyArrayObject*` with the correct type. On failure, the Python error string will be set and the routine returns `NULL`.

**make_contiguous()**

Return type: `PyArrayObject*`

Arguments:

- `PyArrayObject* ary`, a NumPy array.
- `int* is_new_object`, returns a value of 0 if no conversion performed, else 1.
- `int min_dims`, minimum allowable dimensions.
- `int max_dims`, maximum allowable dimensions.

Check to see if `ary` is contiguous. If so, return the input pointer and flag it as not a new object. If it is not contiguous, create a new `PyArrayObject*` using the original data, flag it as a new object and return the pointer.

**make_fortran()**

Return type: `PyArrayObject*`

Arguments

- `PyArrayObject* ary`, a NumPy array.
- `int* is_new_object`, returns a value of 0 if no conversion performed, else 1.

Check to see if `ary` is Fortran contiguous. If so, return the input pointer and flag it as not a new object. If it is not Fortran contiguous, create a new `PyArrayObject*` using the original data, flag it as a new object and return the pointer.

**obj_to_array_contiguous_allow_conversion()**

Return type: `PyArrayObject*`

Arguments:

- `PyObject* input`, a general Python object.
- `int typecode`, the desired NumPy typecode of the resulting array.
- `int* is_new_object`, returns a value of 0 if no conversion performed, else 1.

Convert `input` to a contiguous `PyArrayObject*` of the specified type. If the input object is not a contiguous `PyArrayObject*`, a new one will be created and the new object flag will be set.

**obj_to_array_fortran_allow_conversion()**

Return type: `PyArrayObject*`

Arguments:

- `PyObject* input`, a general Python object.
- `int typecode`, the desired NumPy typecode of the resulting array.
- `int* is_new_object`, returns a value of 0 if no conversion performed, else 1.

Convert `input` to a Fortran contiguous `PyArrayObject*` of the specified type. If the input object is not a Fortran contiguous `PyArrayObject*`, a new one will be created and the new object flag will be set.

**require_contiguous()**

Return type: `int`

Arguments:

- `PyArrayObject* ary`, a NumPy array.

Test whether `ary` is contiguous. If so, return 1. Otherwise, set a Python error and return 0.

**require_native()**

Return type: `int`

Arguments:

- `PyArray_Object* ary`, a NumPy array.

Require that `ary` is not byte-swapped. If the array is not byte-swapped, return 1. Otherwise, set a Python error and return 0.

**require_dimensions()**

Return type: `int`

Arguments:

- `PyArrayObject* ary`, a NumPy array.
- `int exact_dimensions`, the desired number of dimensions.

Require `ary` to have a specified number of dimensions. If the array has the specified number of dimensions, return 1. Otherwise, set a Python error and return 0.

**require_dimensions_n()**

Return type: `int`

Arguments:

- `PyArrayObject* ary`, a NumPy array.
- `int* exact_dimensions`, an array of integers representing acceptable numbers of dimensions.
- `int n`, the length of `exact_dimensions`.

Require `ary` to have one of a list of specified number of dimensions. If the array has one of the specified number of dimensions, return 1. Otherwise, set the Python error string and return 0.

**require_size()**

Return type: `int`

Arguments:

- `PyArrayObject* ary`, a NumPy array.
- `npy_int* size`, an array representing the desired lengths of each dimension.
- `int n`, the length of `size`.

Require `ary` to have a specified shape. If the array has the specified shape, return 1. Otherwise, set the Python error string and return 0.

**require_fortran()**

Return type: `int`

Arguments:

- `PyArrayObject* ary`, a NumPy array.

Require the given `PyArrayObject` to to be Fortran ordered. If the `PyArrayObject` is already Fortran ordered, do nothing. Else, set the Fortran ordering flag and recompute the strides.

## 10.0.6 Beyond the Provided Typemaps

There are many C or C++ array/NumPy array situations not covered by a simple `%include "numpy.i"` and subsequent `%apply` directives.

### A Common Example

Consider a reasonable prototype for a dot product function:

```
double dot(int len, double* vec1, double* vec2);
```

The Python interface that we want is:

```
def dot(vec1, vec2):
    """
    dot(PyObject,PyObject) -> double
    """
```

The problem here is that there is one dimension argument and two array arguments, and our typemaps are set up for dimensions that apply to a single array (in fact, SWIG does not provide a mechanism for associating `len` with `vec2` that takes two Python input arguments). The recommended solution is the following:

```
%apply (int DIM1, double* IN_ARRAY1) {(int len1, double* vec1),
                                      (int len2, double* vec2)}
%rename (dot) my_dot;
%exception my_dot {
    $action
    if (PyErr_Occurred()) SWIG_fail;
}
%inline %{
double my_dot(int len1, double* vec1, int len2, double* vec2) {
    if (len1 != len2) {
        PyErr_Format(PyExc_ValueError,
                     "Arrays of lengths (%d,%d) given",
                     len1, len2);
        return 0.0;
    }
    return dot(len1, vec1, vec2);
}
%}
```

If the header file that contains the prototype for `double dot()` also contains other prototypes that you want to wrap, so that you need to `%include` this header file, then you will also need a `%ignore dot;` directive, placed after the `%rename` and before the `%include` directives. Or, if the function in question is a class method, you will want to use `%extend` rather than `%inline` in addition to `%ignore`.

**A note on error handling:** Note that `my_dot` returns a `double` but that it can also raise a Python error. The resulting wrapper function will return a Python float representation of 0.0 when the vector lengths do not match. Since this is not `NULL`, the Python interpreter will not know to check for an error. For this reason, we add the `%exception` directive above for `my_dot` to get the behavior we want (note that `$action` is a macro that gets expanded to a valid call to `my_dot`). In general, you will probably want to write a SWIG macro to perform this task.

### Other Situations

There are other wrapping situations in which `numpy.i` may be helpful when you encounter them.

- In some situations, it is possible that you could use the `%numpy_typemaps` macro to implement typemaps for your own types. See the *Other Common Types: bool* or *Other Common Types: complex* sections for examples. Another situation is if your dimensions are of a type other than `int` (say `long` for example):

  ```
  %numpy_typemaps(double, NPY_DOUBLE, long)
  ```

- You can use the code in `numpy.i` to write your own typemaps. For example, if you had a five-dimensional array as a function argument, you could cut-and-paste the appropriate four-dimensional typemaps into your interface file. The modifications for the fourth dimension would be trivial.
- Sometimes, the best approach is to use the `%extend` directive to define new methods for your classes (or overload existing ones) that take a `PyObject*` (that either is or can be converted to a `PyArrayObject*`) instead of a pointer to a buffer. In this case, the helper routines in `numpy.i` can be very useful.
- Writing typemaps can be a bit nonintuitive. If you have specific questions about writing SWIG typemaps for NumPy, the developers of `numpy.i` do monitor the Numpy-discussion and Swig-user mail lists.

### A Final Note

When you use the `%apply` directive, as is usually necessary to use `numpy.i`, it will remain in effect until you tell SWIG that it shouldn't be. If the arguments to the functions or methods that you are wrapping have common names, such as `length` or `vector`, these typemaps may get applied in situations you do not expect or want. Therefore, it is always a good idea to add a `%clear` directive after you are done with a specific typemap:

```
%apply (double* IN_ARRAY1, int DIM1) {(double* vector, int length)}
%include "my_header.h"
%clear (double* vector, int length);
```

In general, you should target these typemap signatures specifically where you want them, and then clear them after you are done.

## 10.0.7 Summary

Out of the box, `numpy.i` provides typemaps that support conversion between NumPy arrays and C arrays:

- That can be one of 12 different scalar types: `signed char`, `unsigned char`, `short`, `unsigned short`, `int`, `unsigned int`, `long`, `unsigned long`, `long long`, `unsigned long long`, `float` and `double`.
- That support 74 different argument signatures for each data type, including:
  - One-dimensional, two-dimensional, three-dimensional and four-dimensional arrays.
  - Input-only, in-place, argout, argoutview, and memory managed argoutview behavior.
  - Hard-coded dimensions, data-buffer-then-dimensions specification, and dimensions-then-data-buffer specification.
  - Both C-ordering ("last dimension fastest") or Fortran-ordering ("first dimension fastest") support for 2D, 3D and 4D arrays.

The `numpy.i` interface file also provides additional tools for wrapper developers, including:

- A SWIG macro (`%numpy_typemaps`) with three arguments for implementing the 74 argument signatures for the user's choice of (1) C data type, (2) NumPy data type (assuming they match), and (3) dimension type.
- Fourteen C macros and fifteen C functions that can be used to write specialized typemaps, extensions, or inlined functions that handle cases not covered by the provided typemaps. Note that the macros and functions are coded specifically to work with the NumPy C/API regardless of NumPy version number, both before and after the deprecation of some aspects of the API after version 1.6.

## 10.1 Testing the numpy.i Typemaps

### 10.1.1 Introduction

Writing tests for the `numpy.i` SWIG interface file is a combinatorial headache. At present, 12 different data types are supported, each with 74 different argument signatures, for a total of 888 typemaps supported "out of the box". Each of these typemaps, in turn, might require several unit tests in order to verify expected behavior for both proper and improper inputs. Currently, this results in more than 1,000 individual unit tests executed when `make test` is run in the `numpy/tools/swig` subdirectory.

To facilitate this many similar unit tests, some high-level programming techniques are employed, including C and SWIG macros, as well as Python inheritance. The purpose of this document is to describe the testing infrastructure employed to verify that the `numpy.i` typemaps are working as expected.

### 10.1.2 Testing Organization

There are three independent testing frameworks supported, for one-, two-, and three-dimensional arrays respectively. For one-dimensional arrays, there are two C++ files, a header and a source, named:

```
Vector.h
Vector.cxx
```

that contain prototypes and code for a variety of functions that have one-dimensional arrays as function arguments. The file:

```
Vector.i
```

is a SWIG interface file that defines a python module `Vector` that wraps the functions in `Vector.h` while utilizing the typemaps in `numpy.i` to correctly handle the C arrays.

The `Makefile` calls `swig` to generate `Vector.py` and `Vector_wrap.cxx`, and also executes the `setup.py` script that compiles `Vector_wrap.cxx` and links together the extension module `_Vector.so` or `_Vector.dylib`, depending on the platform. This extension module and the proxy file `Vector.py` are both placed in a subdirectory under the `build` directory.

The actual testing takes place with a Python script named:

```
testVector.py
```

that uses the standard Python library module `unittest`, which performs several tests of each function defined in `Vector.h` for each data type supported.

Two-dimensional arrays are tested in exactly the same manner. The above description applies, but with `Matrix` substituted for `Vector`. For three-dimensional tests, substitute `Tensor` for `Vector`. For four-dimensional tests, substitute `SuperTensor` for `Vector`. For flat in-place array tests, substitute `Flat` for `Vector`. For the descriptions that follow, we will reference the `Vector` tests, but the same information applies to `Matrix`, `Tensor` and `SuperTensor` tests.

The command `make test` will ensure that all of the test software is built and then run all three test scripts.

### 10.1.3 Testing Header Files

`Vector.h` is a C++ header file that defines a C macro called `TEST_FUNC_PROTOS` that takes two arguments: `TYPE`, which is a data type name such as `unsigned int`; and `SNAME`, which is a short name for the same data type with no spaces, e.g. `uint`. This macro defines several function prototypes that have the prefix `SNAME` and have at least one argument that is an array of type `TYPE`. Those functions that have return arguments return a `TYPE` value.

`TEST_FUNC_PROTOS` is then implemented for all of the data types supported by `numpy.i`:

- `signed char`
- `unsigned char`
- `short`
- `unsigned short`
- `int`
- `unsigned int`
- `long`
- `unsigned long`
- `long long`
- `unsigned long long`
- `float`
- `double`

### 10.1.4 Testing Source Files

`Vector.cxx` is a C++ source file that implements compilable code for each of the function prototypes specified in `Vector.h`. It defines a C macro `TEST_FUNCS` that has the same arguments and works in the same way as `TEST_FUNC_PROTOS` does in `Vector.h`. `TEST_FUNCS` is implemented for each of the 12 data types as above.

### 10.1.5 Testing SWIG Interface Files

`Vector.i` is a SWIG interface file that defines python module `Vector`. It follows the conventions for using `numpy.i` as described in this chapter. It defines a SWIG macro `%apply_numpy_typemaps` that has a single argument `TYPE`. It uses the SWIG directive `%apply` to apply the provided typemaps to the argument signatures found in `Vector.h`. This macro is then implemented for all of the data types supported by `numpy.i`. It then does a `%include "Vector.h"` to wrap all of the function prototypes in `Vector.h` using the typemaps in `numpy.i`.

### 10.1.6 Testing Python Scripts

After `make` is used to build the testing extension modules, `testVector.py` can be run to execute the tests. As with other scripts that use `unittest` to facilitate unit testing, `testVector.py` defines a class that inherits from `unittest.TestCase`:

```
class VectorTestCase(unittest.TestCase):
```

However, this class is not run directly. Rather, it serves as a base class to several other python classes, each one specific to a particular data type. The `VectorTestCase` class stores two strings for typing information:

**self.typeStr**

A string that matches one of the `SNAME` prefixes used in `Vector.h` and `Vector.cxx`. For example, `"double"`.

**self.typeCode**

A short (typically single-character) string that represents a data type in numpy and corresponds to `self.typeStr`. For example, if `self.typeStr` is `"double"`, then `self.typeCode` should be `"d"`.

Each test defined by the `VectorTestCase` class extracts the python function it is trying to test by accessing the `Vector` module's dictionary:

```
length = Vector.__dict__[self.typeStr + "Length"]
```

In the case of double precision tests, this will return the python function `Vector.doubleLength`.

We then define a new test case class for each supported data type with a short definition such as:

```
class doubleTestCase(VectorTestCase):
    def __init__(self, methodName="runTest"):
        VectorTestCase.__init__(self, methodName)
        self.typeStr  = "double"
        self.typeCode = "d"
```

Each of these 12 classes is collected into a `unittest.TestSuite`, which is then executed. Errors and failures are summed together and returned as the exit argument. Any non-zero result indicates that at least one test did not pass.

## CHAPTER

## ELEVEN

# ACKNOWLEDGEMENTS

Large parts of this manual originate from Travis E. Oliphant's book Guide to NumPy (which generously entered Public Domain in August 2008). The reference documentation for many of the functions are written by numerous contributors and developers of NumPy.

# BIBLIOGRAPHY

[1] : G. H. Golub and C. F. Van Loan, *Matrix Computations*, 3rd ed., Baltimore, MD, Johns Hopkins University Press, 1996, pg. 8.

[1] : G. H. Golub and C. F. Van Loan, *Matrix Computations*, 3rd ed., Baltimore, MD, Johns Hopkins University Press, 1996, pg. 8.

[1] Wikipedia, “Curve fitting”, https://en.wikipedia.org/wiki/Curve_fitting

[2] Wikipedia, “Polynomial interpolation”, https://en.wikipedia.org/wiki/Polynomial_interpolation

[1] https://en.wikipedia.org/wiki/IEEE_754

[1] Wikipedia, “Two’s complement”, https://en.wikipedia.org/wiki/Two’s_complement

[1] Wikipedia, “Two’s complement”, https://en.wikipedia.org/wiki/Two’s_complement

[Re860718f5533-1] Press, Teukolsky, Vetterling and Flannery, “Numerical Recipes in C++,” 2nd ed, Cambridge University Press, 2002, p. 31.

[CT] Cooley, James W., and John W. Tukey, 1965, “An algorithm for the machine calculation of complex Fourier series,” *Math. Comput.* 19: 297-301.

[CT] Cooley, James W., and John W. Tukey, 1965, “An algorithm for the machine calculation of complex Fourier series,” *Math. Comput.* 19: 297-301.

[NR] Press, W., Teukolsky, S., Vetterline, W.T., and Flannery, B.P., 2007, *Numerical Recipes: The Art of Scientific Computing*, ch. 12-13. Cambridge Univ. Press, Cambridge, UK.

[1] NumPy Enhancement Proposal (NEP) 32, https://numpy.org/neps/nep-0032-remove-financial-functions.html

[2] Wheeler, D. A., E. Rathke, and R. Weir (Eds.) (2009, May). Open Document Format for Office Applications (OpenDocument)v1.2, Part 2: Recalculated Formula (OpenFormula) Format - Annotated Version, Pre-Draft 12. Organization for the Advancement of Structured Information Standards (OASIS). Billerica, MA, USA. [ODT Document]. Available: http://www.oasis-open.org/committees/documents.php?wg_abbrev=office-formula OpenDocument-formula-20090508.odt

[1] NumPy Enhancement Proposal (NEP) 32, https://numpy.org/neps/nep-0032-remove-financial-functions.html

[2] Wheeler, D. A., E. Rathke, and R. Weir (Eds.) (2009, May). Open Document Format for Office Applications (OpenDocument)v1.2, Part 2: Recalculated Formula (OpenFormula) Format - Annotated Version, Pre-Draft 12. Organization for the Advancement of Structured Information Standards (OASIS). Billerica, MA, USA. [ODT Document]. Available: http://www.oasis-open.org/committees/documents.php?wg_abbrev=office-formula OpenDocument-formula-20090508.odt

[1] NumPy Enhancement Proposal (NEP) 32, https://numpy.org/neps/nep-0032-remove-financial-functions.html

[2] L. J. Gitman, "Principles of Managerial Finance, Brief," 3rd ed., Addison-Wesley, 2003, pg. 346.

[1] NumPy Enhancement Proposal (NEP) 32, https://numpy.org/neps/nep-0032-remove-financial-functions.html

[2] Wheeler, D. A., E. Rathke, and R. Weir (Eds.) (2009, May). Open Document Format for Office Applications (OpenDocument)v1.2, Part 2: Recalculated Formula (OpenFormula) Format - Annotated Version, Pre-Draft 12. Organization for the Advancement of Structured Information Standards (OASIS). Billerica, MA, USA. [ODT Document]. Available: http://www.oasis-open.org/committees/documents.php ?wg_abbrev=office-formulaOpenDocument-formula-20090508.odt

[1] NumPy Enhancement Proposal (NEP) 32, https://numpy.org/neps/nep-0032-remove-financial-functions.html

[1] NumPy Enhancement Proposal (NEP) 32, https://numpy.org/neps/nep-0032-remove-financial-functions.html

[1] NumPy Enhancement Proposal (NEP) 32, https://numpy.org/neps/nep-0032-remove-financial-functions.html

[2] L. J. Gitman, "Principles of Managerial Finance, Brief," 3rd ed., Addison-Wesley, 2003, pg. 348.

[1] NumPy Enhancement Proposal (NEP) 32, https://numpy.org/neps/nep-0032-remove-financial-functions.html

[1] NumPy Enhancement Proposal (NEP) 32, https://numpy.org/neps/nep-0032-remove-financial-functions.html

[1] NumPy Enhancement Proposal (NEP) 32, https://numpy.org/neps/nep-0032-remove-financial-functions.html

[2] Wheeler, D. A., E. Rathke, and R. Weir (Eds.) (2009, May). Open Document Format for Office Applications (OpenDocument)v1.2, Part 2: Recalculated Formula (OpenFormula) Format - Annotated Version, Pre-Draft 12. Organization for the Advancement of Structured Information Standards (OASIS). Billerica, MA, USA. [ODT Document]. Available: http://www.oasis-open.org/committees/documents.php?wg_abbrev=office-formula OpenDocument-formula-20090508.odt

[R5cc1f1f25381-1] NumPy Reference, section Generalized Universal Function API.

[1] Format Specification Mini-Language, Python Documentation.

[1] NumPy User Guide, section I/O with NumPy.

[1] Cormen, "Introduction to Algorithms", Chapter 15.2, p. 370-378

[2] https://en.wikipedia.org/wiki/Matrix_chain_multiplication

[1] : G. H. Golub and C. F. Van Loan, *Matrix Computations*, 3rd ed., Baltimore, MD, Johns Hopkins University Press, 1996, pg. 8.

[1] G. Strang, *Linear Algebra and Its Applications*, 2nd Ed., Orlando, FL, Academic Press, Inc., 1980, pg. 222.

[1] G. H. Golub and C. F. Van Loan, *Matrix Computations*, Baltimore, MD, Johns Hopkins University Press, 1985, pg. 15

[1] G. Strang, *Linear Algebra and Its Applications*, Orlando, FL, Academic Press, Inc., 1980, pg. 285.

[1] MATLAB reference documention, "Rank" https://www.mathworks.com/help/techdoc/ref/rank.html

[2] W. H. Press, S. A. Teukolsky, W. T. Vetterling and B. P. Flannery, "Numerical Recipes (3rd edition)", Cambridge University Press, 2007, page 795.

[1] G. Strang, *Linear Algebra and Its Applications*, 2nd Ed., Orlando, FL, Academic Press, Inc., 1980, pg. 22.

[1] G. Strang, *Linear Algebra and Its Applications*, 2nd Ed., Orlando, FL, Academic Press, Inc., 1980, pp. 139-142.

[1] ISO/IEC standard 9899:1999, "Programming language C."

[1] M. Abramowitz and I. A. Stegun, Handbook of Mathematical Functions. New York, NY: Dover, 1972, pg. 83. http://www.math.sfu.ca/~cbm/aands/

[2] Wikipedia, "Hyperbolic function", https://en.wikipedia.org/wiki/Hyperbolic_function

[1] M. Abramowitz and I.A. Stegun, "Handbook of Mathematical Functions", 10th printing, 1964, pp. 86. http://www.math.sfu.ca/~cbm/aands/

[2] Wikipedia, "Inverse hyperbolic function", https://en.wikipedia.org/wiki/Arcsinh

[1] M. Abramowitz and I.A. Stegun, "Handbook of Mathematical Functions", 10th printing, 1964, pp. 86. http://www.math.sfu.ca/~cbm/aands/

[2] Wikipedia, "Inverse hyperbolic function", https://en.wikipedia.org/wiki/Arccosh

[1] M. Abramowitz and I.A. Stegun, "Handbook of Mathematical Functions", 10th printing, 1964, pp. 86. http://www.math.sfu.ca/~cbm/aands/

[2] Wikipedia, "Inverse hyperbolic function", https://en.wikipedia.org/wiki/Arctanh

[1] "Lecture Notes on the Status of IEEE 754", William Kahan, https://people.eecs.berkeley.edu/~wkahan/ieee754status/IEEE754.PDF

[2] "How Futile are Mindless Assessments of Roundoff in Floating-Point Computation?", William Kahan, https://people.eecs.berkeley.edu/~wkahan/Mindless.pdf

[1] Quarteroni A., Sacco R., Saleri F. (2007) Numerical Mathematics (Texts in Applied Mathematics). New York: Springer.

[2] Durran D. R. (1999) Numerical Methods for Wave Equations in Geophysical Fluid Dynamics. New York: Springer.

[3] Fornberg B. (1988) Generation of Finite Difference Formulas on Arbitrarily Spaced Grids, Mathematics of Computation 51, no. 184 : 699-706. PDF.

[1] Wikipedia page: https://en.wikipedia.org/wiki/Trapezoidal_rule

[2] Illustration image: https://en.wikipedia.org/wiki/File:Composite_trapezoidal_rule_illustration.png

[1] Wikipedia, "Exponential function", https://en.wikipedia.org/wiki/Exponential_function

[2] M. Abramovitz and I. A. Stegun, "Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables," Dover, 1964, p. 69, http://www.math.sfu.ca/~cbm/aands/page_69.htm

[1] M. Abramowitz and I.A. Stegun, "Handbook of Mathematical Functions", 10th printing, 1964, pp. 67. http://www.math.sfu.ca/~cbm/aands/

[2] Wikipedia, "Logarithm". https://en.wikipedia.org/wiki/Logarithm

[1] M. Abramowitz and I.A. Stegun, "Handbook of Mathematical Functions", 10th printing, 1964, pp. 67. http://www.math.sfu.ca/~cbm/aands/

[2] Wikipedia, "Logarithm". https://en.wikipedia.org/wiki/Logarithm

[1] M. Abramowitz and I.A. Stegun, "Handbook of Mathematical Functions", 10th printing, 1964, pp. 67. http://www.math.sfu.ca/~cbm/aands/

[2] Wikipedia, "Logarithm". https://en.wikipedia.org/wiki/Logarithm

[1] C. W. Clenshaw, “Chebyshev series for mathematical functions”, in *National Physical Laboratory Mathematical Tables*, vol. 5, London: Her Majesty’s Stationery Office, 1962.

[2] M. Abramowitz and I. A. Stegun, *Handbook of Mathematical Functions*, 10th printing, New York: Dover, 1964, pp. 379. http://www.math.sfu.ca/~cbm/aands/page_379.htm

[3] http://kobesearch.cpan.org/htdocs/Math-Cephes/Math/Cephes.html

[1] Weisstein, Eric W. “Sinc Function.” From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/SincFunction.html

[2] Wikipedia, “Sinc function”, https://en.wikipedia.org/wiki/Sinc_function

[1] Wikipedia, “Convolution”, https://en.wikipedia.org/wiki/Convolution

[1] Wikipedia, “Curve fitting”, https://en.wikipedia.org/wiki/Curve_fitting

[1] A. T. Benjamin, et al., “Combinatorial Trigonometry with Chebyshev Polynomials,” *Journal of Statistical Planning and Inference 14*, 2008 (preprint: https://www.math.hmc.edu/~benjamin/papers/CombTrig.pdf, pg. 4)

[1] Wikipedia, “Curve fitting”, https://en.wikipedia.org/wiki/Curve_fitting

[1] Wikipedia, “Curve fitting”, https://en.wikipedia.org/wiki/Curve_fitting

[1] Wikipedia, “Curve fitting”, https://en.wikipedia.org/wiki/Curve_fitting

[1] Wikipedia, “Curve fitting”, https://en.wikipedia.org/wiki/Curve_fitting

[1] I. N. Bronshtein, K. A. Semendyayev, and K. A. Hirsch (Eng. trans. Ed.), *Handbook of Mathematics*, New York, Van Nostrand Reinhold Co., 1985, pg. 720.

[1] M. Sullivan and M. Sullivan, III, “Algebra and Trignometry, Enhanced With Graphing Utilities,” Prentice-Hall, pg. 318, 1996.

[2] G. Strang, “Linear Algebra and Its Applications, 2nd Edition,” Academic Press, pg. 182, 1980.

[1] R. A. Horn & C. R. Johnson, *Matrix Analysis*. Cambridge, UK: Cambridge University Press, 1999, pp. 146-7.

[1] Wikipedia, “Curve fitting”, https://en.wikipedia.org/wiki/Curve_fitting

[2] Wikipedia, “Polynomial interpolation”, https://en.wikipedia.org/wiki/Polynomial_interpolation

[1] Daniel Lemire., “Fast Random Integer Generation in an Interval”, ACM Transactions on Modeling and Computer Simulation 29 (1), 2019, http://arxiv.org/abs/1805.10941.

[1] Dalgaard, Peter, “Introductory Statistics with R”, Springer-Verlag, 2002.

[2] Glantz, Stanton A. “Primer of Biostatistics.”, McGraw-Hill, Fifth Edition, 2002.

[3] Lentner, Marvin, “Elementary Applied Statistics”, Bogden and Quigley, 1972.

[4] Weisstein, Eric W. “Binomial Distribution.” From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/BinomialDistribution.html

[5] Wikipedia, “Binomial distribution”, https://en.wikipedia.org/wiki/Binomial_distribution

[1] NIST “Engineering Statistics Handbook” https://www.itl.nist.gov/div898/handbook/eda/section3/eda3666.htm

[1] David McKay, “Information Theory, Inference and Learning Algorithms,” chapter 23, http://www.inference.org.uk/mackay/itila/

[2] Wikipedia, “Dirichlet distribution”, https://en.wikipedia.org/wiki/Dirichlet_distribution

[1] Peyton Z. Peebles Jr., “Probability, Random Variables and Random Signal Principles”, 4th ed, 2001, p. 57.

[2] Wikipedia, “Poisson process”, https://en.wikipedia.org/wiki/Poisson_process

[3] Wikipedia, “Exponential distribution”, https://en.wikipedia.org/wiki/Exponential_distribution

[1] Glantz, Stanton A. “Primer of Biostatistics.”, McGraw-Hill, Fifth Edition, 2002.

[2] Wikipedia, “F-distribution”, https://en.wikipedia.org/wiki/F-distribution

[1] Weisstein, Eric W. “Gamma Distribution.” From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/GammaDistribution.html

[2] Wikipedia, “Gamma distribution”, https://en.wikipedia.org/wiki/Gamma_distribution

[1] Gumbel, E. J., “Statistics of Extremes,” New York: Columbia University Press, 1958.

[2] Reiss, R.-D. and Thomas, M., “Statistical Analysis of Extreme Values from Insurance, Finance, Hydrology and Other Fields,” Basel: Birkhauser Verlag, 2001.

[1] Lentner, Marvin, “Elementary Applied Statistics”, Bogden and Quigley, 1972.

[2] Weisstein, Eric W. “Hypergeometric Distribution.” From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/HypergeometricDistribution.html

[3] Wikipedia, “Hypergeometric distribution”, https://en.wikipedia.org/wiki/Hypergeometric_distribution

[4] Stadlober, Ernst, “The ratio of uniforms approach for generating discrete random variates”, Journal of Computational and Applied Mathematics, 31, pp. 181-189 (1990).

[1] Abramowitz, M. and Stegun, I. A. (Eds.). “Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables, 9th printing,” New York: Dover, 1972.

[2] Kotz, Samuel, et. al. “The Laplace Distribution and Generalizations, ” Birkhauser, 2001.

[3] Weisstein, Eric W. “Laplace Distribution.” From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/LaplaceDistribution.html

[4] Wikipedia, “Laplace distribution”, https://en.wikipedia.org/wiki/Laplace_distribution

[1] Reiss, R.-D. and Thomas M. (2001), “Statistical Analysis of Extreme Values, from Insurance, Finance, Hydrology and Other Fields,” Birkhauser Verlag, Basel, pp 132-133.

[2] Weisstein, Eric W. “Logistic Distribution.” From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/LogisticDistribution.html

[3] Wikipedia, “Logistic-distribution”, https://en.wikipedia.org/wiki/Logistic_distribution

[1] Limpert, E., Stahel, W. A., and Abbt, M., “Log-normal Distributions across the Sciences: Keys and Clues,” BioScience, Vol. 51, No. 5, May, 2001. https://stat.ethz.ch/~stahel/lognormal/bioscience.pdf

[2] Reiss, R.D. and Thomas, M., “Statistical Analysis of Extreme Values,” Basel: Birkhauser Verlag, 2001, pp. 31-32.

[1] Buzas, Martin A.; Culver, Stephen J., Understanding regional species diversity through the log series distribution of occurrences: BIODIVERSITY RESEARCH Diversity & Distributions, Volume 5, Number 5, September 1999 , pp. 187-195(9).

[2] Fisher, R.A„ A.S. Corbet, and C.B. Williams. 1943. The relation between the number of species and the number of individuals in a random sample of an animal population. Journal of Animal Ecology, 12:42-58.

[3] D. J. Hand, F. Daly, D. Lunn, E. Ostrowski, A Handbook of Small Data Sets, CRC Press, 1994.

[4] Wikipedia, “Logarithmic distribution”, https://en.wikipedia.org/wiki/Logarithmic_distribution

[1] Papoulis, A., “Probability, Random Variables, and Stochastic Processes,” 3rd ed., New York: McGraw-Hill, 1991.

[2] Duda, R. O., Hart, P. E., and Stork, D. G., “Pattern Classification,” 2nd ed., New York: Wiley, 2001.

[1] Weisstein, Eric W. "Negative Binomial Distribution." From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/NegativeBinomialDistribution.html

[2] Wikipedia, "Negative binomial distribution", https://en.wikipedia.org/wiki/Negative_binomial_distribution

[1] Wikipedia, "Noncentral chi-squared distribution" https://en.wikipedia.org/wiki/Noncentral_chi-squared_distribution

[1] Weisstein, Eric W. "Noncentral F-Distribution." From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/NoncentralF-Distribution.html

[2] Wikipedia, "Noncentral F-distribution", https://en.wikipedia.org/wiki/Noncentral_F-distribution

[1] Wikipedia, "Normal distribution", https://en.wikipedia.org/wiki/Normal_distribution

[2] P. R. Peebles Jr., "Central Limit Theorem" in "Probability, Random Variables and Random Signal Principles", 4th ed., 2001, pp. 51, 51, 125.

[1] Francis Hunt and Paul Johnson, On the Pareto Distribution of Sourceforge projects.

[2] Pareto, V. (1896). Course of Political Economy. Lausanne.

[3] Reiss, R.D., Thomas, M.(2001), Statistical Analysis of Extreme Values, Birkhauser Verlag, Basel, pp 23-30.

[4] Wikipedia, "Pareto distribution", https://en.wikipedia.org/wiki/Pareto_distribution

[1] Weisstein, Eric W. "Poisson Distribution." From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/PoissonDistribution.html

[2] Wikipedia, "Poisson distribution", https://en.wikipedia.org/wiki/Poisson_distribution

[1] Christian Kleiber, Samuel Kotz, "Statistical size distributions in economics and actuarial sciences", Wiley, 2003.

[2] Heckert, N. A. and Filliben, James J. "NIST Handbook 148: Dataplot Reference Manual, Volume 2: Let Subcommands and Library Functions", National Institute of Standards and Technology Handbook Series, June 2003. https://www.itl.nist.gov/div898/software/dataplot/refman2/auxillar/powpdf.pdf

[1] Brighton Webs Ltd., "Rayleigh Distribution," https://web.archive.org/web/20090514091424/http://brighton-webs.co.uk:80/distributions/rayleigh.asp

[2] Wikipedia, "Rayleigh distribution" https://en.wikipedia.org/wiki/Rayleigh_distribution

[1] NIST/SEMATECH e-Handbook of Statistical Methods, "Cauchy Distribution", https://www.itl.nist.gov/div898/handbook/eda/section3/eda3663.htm

[2] Weisstein, Eric W. "Cauchy Distribution." From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/CauchyDistribution.html

[3] Wikipedia, "Cauchy distribution" https://en.wikipedia.org/wiki/Cauchy_distribution

[1] Weisstein, Eric W. "Gamma Distribution." From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/GammaDistribution.html

[2] Wikipedia, "Gamma distribution", https://en.wikipedia.org/wiki/Gamma_distribution

[1] Dalgaard, Peter, "Introductory Statistics With R", Springer, 2002.

[2] Wikipedia, "Student's t-distribution" https://en.wikipedia.org/wiki/Student's_t-distribution

[1] Wikipedia, "Triangular distribution" https://en.wikipedia.org/wiki/Triangular_distribution

[1] Abramowitz, M. and Stegun, I. A. (Eds.). "Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables, 9th printing," New York: Dover, 1972.

[2] von Mises, R., "Mathematical Theory of Probability and Statistics", New York: Academic Press, 1964.

[1] Brighton Webs Ltd., Wald Distribution, https://web.archive.org/web/20090423014010/http://www.brighton-webs.co.uk:80/distributions/wald.asp

[2] Chhikara, Raj S., and Folks, J. Leroy, "The Inverse Gaussian Distribution: Theory : Methodology, and Applications", CRC Press, 1988.

[3] Wikipedia, "Inverse Gaussian distribution" https://en.wikipedia.org/wiki/Inverse_Gaussian_distribution

[1] Waloddi Weibull, Royal Technical University, Stockholm, 1939 "A Statistical Theory Of The Strength Of Materials", Ingeniorsvetenskapsakademiens Handlingar Nr 151, 1939, Generalstabens Litografiska Anstalts Forlag, Stockholm.

[2] Waloddi Weibull, "A Statistical Distribution Function of Wide Applicability", Journal Of Applied Mechanics ASME Paper 1951.

[3] Wikipedia, "Weibull distribution", https://en.wikipedia.org/wiki/Weibull_distribution

[1] Zipf, G. K., "Selected Studies of the Principle of Relative Frequency in Language," Cambridge, MA: Harvard Univ. Press, 1932.

[1] M. Matsumoto and T. Nishimura, "Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator," *ACM Trans. on Modeling and Computer Simulation*, Vol. 8, No. 1, pp. 3-30, Jan. 1998.

[1] Dalgaard, Peter, "Introductory Statistics with R", Springer-Verlag, 2002.

[2] Glantz, Stanton A. "Primer of Biostatistics.", McGraw-Hill, Fifth Edition, 2002.

[3] Lentner, Marvin, "Elementary Applied Statistics", Bogden and Quigley, 1972.

[4] Weisstein, Eric W. "Binomial Distribution." From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/BinomialDistribution.html

[5] Wikipedia, "Binomial distribution", https://en.wikipedia.org/wiki/Binomial_distribution

[1] NIST "Engineering Statistics Handbook" https://www.itl.nist.gov/div898/handbook/eda/section3/eda3666.htm

[1] David McKay, "Information Theory, Inference and Learning Algorithms," chapter 23, http://www.inference.org.uk/mackay/itila/

[2] Wikipedia, "Dirichlet distribution", https://en.wikipedia.org/wiki/Dirichlet_distribution

[1] Peyton Z. Peebles Jr., "Probability, Random Variables and Random Signal Principles", 4th ed, 2001, p. 57.

[2] Wikipedia, "Poisson process", https://en.wikipedia.org/wiki/Poisson_process

[3] Wikipedia, "Exponential distribution", https://en.wikipedia.org/wiki/Exponential_distribution

[1] Glantz, Stanton A. "Primer of Biostatistics.", McGraw-Hill, Fifth Edition, 2002.

[2] Wikipedia, "F-distribution", https://en.wikipedia.org/wiki/F-distribution

[1] Weisstein, Eric W. "Gamma Distribution." From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/GammaDistribution.html

[2] Wikipedia, "Gamma distribution", https://en.wikipedia.org/wiki/Gamma_distribution

[1] Gumbel, E. J., "Statistics of Extremes," New York: Columbia University Press, 1958.

[2] Reiss, R.-D. and Thomas, M., "Statistical Analysis of Extreme Values from Insurance, Finance, Hydrology and Other Fields," Basel: Birkhauser Verlag, 2001.

[1] Lentner, Marvin, "Elementary Applied Statistics", Bogden and Quigley, 1972.

[2] Weisstein, Eric W. "Hypergeometric Distribution." From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/HypergeometricDistribution.html

[3] Wikipedia, “Hypergeometric distribution”, https://en.wikipedia.org/wiki/Hypergeometric_distribution

[1] Abramowitz, M. and Stegun, I. A. (Eds.). “Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables, 9th printing,” New York: Dover, 1972.

[2] Kotz, Samuel, et. al. “The Laplace Distribution and Generalizations, ” Birkhauser, 2001.

[3] Weisstein, Eric W. “Laplace Distribution.” From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/LaplaceDistribution.html

[4] Wikipedia, “Laplace distribution”, https://en.wikipedia.org/wiki/Laplace_distribution

[1] Reiss, R.-D. and Thomas M. (2001), “Statistical Analysis of Extreme Values, from Insurance, Finance, Hydrology and Other Fields,” Birkhauser Verlag, Basel, pp 132-133.

[2] Weisstein, Eric W. “Logistic Distribution.” From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/LogisticDistribution.html

[3] Wikipedia, “Logistic-distribution”, https://en.wikipedia.org/wiki/Logistic_distribution

[1] Limpert, E., Stahel, W. A., and Abbt, M., “Log-normal Distributions across the Sciences: Keys and Clues,” BioScience, Vol. 51, No. 5, May, 2001. https://stat.ethz.ch/~stahel/lognormal/bioscience.pdf

[2] Reiss, R.D. and Thomas, M., “Statistical Analysis of Extreme Values,” Basel: Birkhauser Verlag, 2001, pp. 31-32.

[1] Buzas, Martin A.; Culver, Stephen J., Understanding regional species diversity through the log series distribution of occurrences: BIODIVERSITY RESEARCH Diversity & Distributions, Volume 5, Number 5, September 1999 , pp. 187-195(9).

[2] Fisher, R.A„ A.S. Corbet, and C.B. Williams. 1943. The relation between the number of species and the number of individuals in a random sample of an animal population. Journal of Animal Ecology, 12:42-58.

[3] D. J. Hand, F. Daly, D. Lunn, E. Ostrowski, A Handbook of Small Data Sets, CRC Press, 1994.

[4] Wikipedia, “Logarithmic distribution”, https://en.wikipedia.org/wiki/Logarithmic_distribution

[1] Papoulis, A., “Probability, Random Variables, and Stochastic Processes,” 3rd ed., New York: McGraw-Hill, 1991.

[2] Duda, R. O., Hart, P. E., and Stork, D. G., “Pattern Classification,” 2nd ed., New York: Wiley, 2001.

[1] Weisstein, Eric W. “Negative Binomial Distribution.” From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/NegativeBinomialDistribution.html

[2] Wikipedia, “Negative binomial distribution”, https://en.wikipedia.org/wiki/Negative_binomial_distribution

[1] Wikipedia, “Noncentral chi-squared distribution” https://en.wikipedia.org/wiki/Noncentral_chi-squared_distribution

[1] Weisstein, Eric W. “Noncentral F-Distribution.” From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/NoncentralF-Distribution.html

[2] Wikipedia, “Noncentral F-distribution”, https://en.wikipedia.org/wiki/Noncentral_F-distribution

[1] Wikipedia, “Normal distribution”, https://en.wikipedia.org/wiki/Normal_distribution

[2] P. R. Peebles Jr., “Central Limit Theorem” in “Probability, Random Variables and Random Signal Principles”, 4th ed., 2001, pp. 51, 51, 125.

[1] Francis Hunt and Paul Johnson, On the Pareto Distribution of Sourceforge projects.

[2] Pareto, V. (1896). Course of Political Economy. Lausanne.

[3] Reiss, R.D., Thomas, M.(2001), Statistical Analysis of Extreme Values, Birkhauser Verlag, Basel, pp 23-30.

[4] Wikipedia, “Pareto distribution”, https://en.wikipedia.org/wiki/Pareto_distribution

[1] Weisstein, Eric W. “Poisson Distribution.” From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/PoissonDistribution.html

[2] Wikipedia, “Poisson distribution”, https://en.wikipedia.org/wiki/Poisson_distribution

[1] Christian Kleiber, Samuel Kotz, “Statistical size distributions in economics and actuarial sciences”, Wiley, 2003.

[2] Heckert, N. A. and Filliben, James J. “NIST Handbook 148: Dataplot Reference Manual, Volume 2: Let Subcommands and Library Functions”, National Institute of Standards and Technology Handbook Series, June 2003. https://www.itl.nist.gov/div898/software/dataplot/refman2/auxillar/powpdf.pdf

[1] Brighton Webs Ltd., “Rayleigh Distribution,” https://web.archive.org/web/20090514091424/http://brighton-webs.co.uk:80/distributions/rayleigh.asp

[2] Wikipedia, “Rayleigh distribution” https://en.wikipedia.org/wiki/Rayleigh_distribution

[1] NIST/SEMATECH e-Handbook of Statistical Methods, “Cauchy Distribution”, https://www.itl.nist.gov/div898/handbook/eda/section3/eda3663.htm

[2] Weisstein, Eric W. “Cauchy Distribution.” From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/CauchyDistribution.html

[3] Wikipedia, “Cauchy distribution” https://en.wikipedia.org/wiki/Cauchy_distribution

[1] Weisstein, Eric W. “Gamma Distribution.” From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/GammaDistribution.html

[2] Wikipedia, “Gamma distribution”, https://en.wikipedia.org/wiki/Gamma_distribution

[1] Dalgaard, Peter, “Introductory Statistics With R”, Springer, 2002.

[2] Wikipedia, “Student’s t-distribution” https://en.wikipedia.org/wiki/Student’s_t-distribution

[1] Wikipedia, “Triangular distribution” https://en.wikipedia.org/wiki/Triangular_distribution

[1] Abramowitz, M. and Stegun, I. A. (Eds.). “Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables, 9th printing,” New York: Dover, 1972.

[2] von Mises, R., “Mathematical Theory of Probability and Statistics”, New York: Academic Press, 1964.

[1] Brighton Webs Ltd., Wald Distribution, https://web.archive.org/web/20090423014010/http://www.brighton-webs.co.uk:80/distributions/wald.asp

[2] Chhikara, Raj S., and Folks, J. Leroy, “The Inverse Gaussian Distribution: Theory : Methodology, and Applications”, CRC Press, 1988.

[3] Wikipedia, “Inverse Gaussian distribution” https://en.wikipedia.org/wiki/Inverse_Gaussian_distribution

[1] Waloddi Weibull, Royal Technical University, Stockholm, 1939 “A Statistical Theory Of The Strength Of Materials”, Ingeniorsvetenskapsakademiens Handlingar Nr 151, 1939, Generalstabens Litografiska Anstalts Forlag, Stockholm.

[2] Waloddi Weibull, “A Statistical Distribution Function of Wide Applicability”, Journal Of Applied Mechanics ASME Paper 1951.

[3] Wikipedia, “Weibull distribution”, https://en.wikipedia.org/wiki/Weibull_distribution

[1] Zipf, G. K., “Selected Studies of the Principle of Relative Frequency in Language,” Cambridge, MA: Harvard Univ. Press, 1932.

[1] Dalgaard, Peter, “Introductory Statistics with R”, Springer-Verlag, 2002.

[2] Glantz, Stanton A. “Primer of Biostatistics.”, McGraw-Hill, Fifth Edition, 2002.

[3] Lentner, Marvin, "Elementary Applied Statistics", Bogden and Quigley, 1972.

[4] Weisstein, Eric W. "Binomial Distribution." From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/BinomialDistribution.html

[5] Wikipedia, "Binomial distribution", https://en.wikipedia.org/wiki/Binomial_distribution

[1] NIST "Engineering Statistics Handbook" https://www.itl.nist.gov/div898/handbook/eda/section3/eda3666.htm

[1] David McKay, "Information Theory, Inference and Learning Algorithms," chapter 23, http://www.inference.org.uk/mackay/itila/

[2] Wikipedia, "Dirichlet distribution", https://en.wikipedia.org/wiki/Dirichlet_distribution

[1] Peyton Z. Peebles Jr., "Probability, Random Variables and Random Signal Principles", 4th ed, 2001, p. 57.

[2] Wikipedia, "Poisson process", https://en.wikipedia.org/wiki/Poisson_process

[3] Wikipedia, "Exponential distribution", https://en.wikipedia.org/wiki/Exponential_distribution

[1] Glantz, Stanton A. "Primer of Biostatistics.", McGraw-Hill, Fifth Edition, 2002.

[2] Wikipedia, "F-distribution", https://en.wikipedia.org/wiki/F-distribution

[1] Weisstein, Eric W. "Gamma Distribution." From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/GammaDistribution.html

[2] Wikipedia, "Gamma distribution", https://en.wikipedia.org/wiki/Gamma_distribution

[1] Gumbel, E. J., "Statistics of Extremes," New York: Columbia University Press, 1958.

[2] Reiss, R.-D. and Thomas, M., "Statistical Analysis of Extreme Values from Insurance, Finance, Hydrology and Other Fields," Basel: Birkhauser Verlag, 2001.

[1] Lentner, Marvin, "Elementary Applied Statistics", Bogden and Quigley, 1972.

[2] Weisstein, Eric W. "Hypergeometric Distribution." From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/HypergeometricDistribution.html

[3] Wikipedia, "Hypergeometric distribution", https://en.wikipedia.org/wiki/Hypergeometric_distribution

[1] Abramowitz, M. and Stegun, I. A. (Eds.). "Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables, 9th printing," New York: Dover, 1972.

[2] Kotz, Samuel, et. al. "The Laplace Distribution and Generalizations, " Birkhauser, 2001.

[3] Weisstein, Eric W. "Laplace Distribution." From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/LaplaceDistribution.html

[4] Wikipedia, "Laplace distribution", https://en.wikipedia.org/wiki/Laplace_distribution

[1] Reiss, R.-D. and Thomas M. (2001), "Statistical Analysis of Extreme Values, from Insurance, Finance, Hydrology and Other Fields," Birkhauser Verlag, Basel, pp 132-133.

[2] Weisstein, Eric W. "Logistic Distribution." From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/LogisticDistribution.html

[3] Wikipedia, "Logistic-distribution", https://en.wikipedia.org/wiki/Logistic_distribution

[1] Limpert, E., Stahel, W. A., and Abbt, M., "Log-normal Distributions across the Sciences: Keys and Clues," BioScience, Vol. 51, No. 5, May, 2001. https://stat.ethz.ch/~stahel/lognormal/bioscience.pdf

[2] Reiss, R.D. and Thomas, M., "Statistical Analysis of Extreme Values," Basel: Birkhauser Verlag, 2001, pp. 31-32.

[1] Buzas, Martin A.; Culver, Stephen J., Understanding regional species diversity through the log series distribution of occurrences: BIODIVERSITY RESEARCH Diversity & Distributions, Volume 5, Number 5, September 1999 , pp. 187-195(9).

[2] Fisher, R.A„ A.S. Corbet, and C.B. Williams. 1943. The relation between the number of species and the number of individuals in a random sample of an animal population. Journal of Animal Ecology, 12:42-58.

[3] D. J. Hand, F. Daly, D. Lunn, E. Ostrowski, A Handbook of Small Data Sets, CRC Press, 1994.

[4] Wikipedia, “Logarithmic distribution”, https://en.wikipedia.org/wiki/Logarithmic_distribution

[1] Papoulis, A., “Probability, Random Variables, and Stochastic Processes,” 3rd ed., New York: McGraw-Hill, 1991.

[2] Duda, R. O., Hart, P. E., and Stork, D. G., “Pattern Classification,” 2nd ed., New York: Wiley, 2001.

[1] Weisstein, Eric W. “Negative Binomial Distribution.” From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/NegativeBinomialDistribution.html

[2] Wikipedia, “Negative binomial distribution”, https://en.wikipedia.org/wiki/Negative_binomial_distribution

[1] Wikipedia, “Noncentral chi-squared distribution” https://en.wikipedia.org/wiki/Noncentral_chi-squared_distribution

[1] Weisstein, Eric W. “Noncentral F-Distribution.” From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/NoncentralF-Distribution.html

[2] Wikipedia, “Noncentral F-distribution”, https://en.wikipedia.org/wiki/Noncentral_F-distribution

[1] Wikipedia, “Normal distribution”, https://en.wikipedia.org/wiki/Normal_distribution

[2] P. R. Peebles Jr., “Central Limit Theorem” in “Probability, Random Variables and Random Signal Principles”, 4th ed., 2001, pp. 51, 51, 125.

[1] Francis Hunt and Paul Johnson, On the Pareto Distribution of Sourceforge projects.

[2] Pareto, V. (1896). Course of Political Economy. Lausanne.

[3] Reiss, R.D., Thomas, M.(2001), Statistical Analysis of Extreme Values, Birkhauser Verlag, Basel, pp 23-30.

[4] Wikipedia, “Pareto distribution”, https://en.wikipedia.org/wiki/Pareto_distribution

[1] Weisstein, Eric W. “Poisson Distribution.” From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/PoissonDistribution.html

[2] Wikipedia, “Poisson distribution”, https://en.wikipedia.org/wiki/Poisson_distribution

[1] Christian Kleiber, Samuel Kotz, “Statistical size distributions in economics and actuarial sciences”, Wiley, 2003.

[2] Heckert, N. A. and Filliben, James J. “NIST Handbook 148: Dataplot Reference Manual, Volume 2: Let Subcommands and Library Functions”, National Institute of Standards and Technology Handbook Series, June 2003. https://www.itl.nist.gov/div898/software/dataplot/refman2/auxillar/powpdf.pdf

[1] Brighton Webs Ltd., “Rayleigh Distribution,” https://web.archive.org/web/20090514091424/http://brighton-webs.co.uk:80/distributions/rayleigh.asp

[2] Wikipedia, “Rayleigh distribution” https://en.wikipedia.org/wiki/Rayleigh_distribution

[1] M. Matsumoto and T. Nishimura, “Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator,” *ACM Trans. on Modeling and Computer Simulation*, Vol. 8, No. 1, pp. 3-30, Jan. 1998.

[1] NIST/SEMATECH e-Handbook of Statistical Methods, “Cauchy Distribution”, https://www.itl.nist.gov/div898/handbook/eda/section3/eda3663.htm

[2] Weisstein, Eric W. "Cauchy Distribution." From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/CauchyDistribution.html

[3] Wikipedia, "Cauchy distribution" https://en.wikipedia.org/wiki/Cauchy_distribution

[1] Weisstein, Eric W. "Gamma Distribution." From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/GammaDistribution.html

[2] Wikipedia, "Gamma distribution", https://en.wikipedia.org/wiki/Gamma_distribution

[1] Dalgaard, Peter, "Introductory Statistics With R", Springer, 2002.

[2] Wikipedia, "Student's t-distribution" https://en.wikipedia.org/wiki/Student's_t-distribution

[1] Wikipedia, "Triangular distribution" https://en.wikipedia.org/wiki/Triangular_distribution

[1] Abramowitz, M. and Stegun, I. A. (Eds.). "Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables, 9th printing," New York: Dover, 1972.

[2] von Mises, R., "Mathematical Theory of Probability and Statistics", New York: Academic Press, 1964.

[1] Brighton Webs Ltd., Wald Distribution, https://web.archive.org/web/20090423014010/http://www.brighton-webs.co.uk:80/distributions/wald.asp

[2] Chhikara, Raj S., and Folks, J. Leroy, "The Inverse Gaussian Distribution: Theory : Methodology, and Applications", CRC Press, 1988.

[3] Wikipedia, "Inverse Gaussian distribution" https://en.wikipedia.org/wiki/Inverse_Gaussian_distribution

[1] Waloddi Weibull, Royal Technical University, Stockholm, 1939 "A Statistical Theory Of The Strength Of Materials", Ingeniorsvetenskapsakademiens Handlingar Nr 151, 1939, Generalstabens Litografiska Anstalts Forlag, Stockholm.

[2] Waloddi Weibull, "A Statistical Distribution Function of Wide Applicability", Journal Of Applied Mechanics ASME Paper 1951.

[3] Wikipedia, "Weibull distribution", https://en.wikipedia.org/wiki/Weibull_distribution

[1] Zipf, G. K., "Selected Studies of the Principle of Relative Frequency in Language," Cambridge, MA: Harvard Univ. Press, 1932.

[1] Hiroshi Haramoto, Makoto Matsumoto, and Pierre L'Ecuyer, "A Fast Jump Ahead Algorithm for Linear Recurrences in a Polynomial Space", Sequences and Their Applications - SETA, 290–298, 2008.

[2] Hiroshi Haramoto, Makoto Matsumoto, Takuji Nishimura, François Panneton, Pierre L'Ecuyer, "Efficient Jump Ahead for F2-Linear Random Number Generators", INFORMS JOURNAL ON COMPUTING, Vol. 20, No. 3, Summer 2008, pp. 385-390.

[1] Matsumoto, M, Generating multiple disjoint streams of pseudorandom number sequences. Accessed on: May 6, 2020. http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/JUMP/

[2] Hiroshi Haramoto, Makoto Matsumoto, Takuji Nishimura, François Panneton, Pierre L'Ecuyer, "Efficient Jump Ahead for F2-Linear Random Number Generators", INFORMS JOURNAL ON COMPUTING, Vol. 20, No. 3, Summer 2008, pp. 385-390.

[1] "PCG, A Family of Better Random Number Generators"

[2] O'Neill, Melissa E. "PCG: A Family of Simple Fast Space-Efficient Statistically Good Algorithms for Random Number Generation"

[1] John K. Salmon, Mark A. Moraes, Ron O. Dror, and David E. Shaw, "Parallel Random Numbers: As Easy as 1, 2, 3," Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC11), New York, NY: ACM, 2011.

[1] "PractRand"

[2] “Random Invertible Mapping Statistics”

[1] M.S. Bartlett, “Periodogram Analysis and Continuous Spectra”, Biometrika 37, 1-16, 1950.

[2] E.R. Kanasewich, “Time Sequence Analysis in Geophysics”, The University of Alberta Press, 1975, pp. 109-110.

[3] A.V. Oppenheim and R.W. Schafer, “Discrete-Time Signal Processing”, Prentice-Hall, 1999, pp. 468-471.

[4] Wikipedia, “Window function”, https://en.wikipedia.org/wiki/Window_function

[5] W.H. Press, B.P. Flannery, S.A. Teukolsky, and W.T. Vetterling, “Numerical Recipes”, Cambridge University Press, 1986, page 429.

[1] Blackman, R.B. and Tukey, J.W., (1958) The measurement of power spectra, Dover Publications, New York.

[2] E.R. Kanasewich, “Time Sequence Analysis in Geophysics”, The University of Alberta Press, 1975, pp. 109-110.

[3] Wikipedia, “Window function”, https://en.wikipedia.org/wiki/Window_function

[4] W.H. Press, B.P. Flannery, S.A. Teukolsky, and W.T. Vetterling, “Numerical Recipes”, Cambridge University Press, 1986, page 425.

[1] Blackman, R.B. and Tukey, J.W., (1958) The measurement of power spectra, Dover Publications, New York.

[2] E.R. Kanasewich, “Time Sequence Analysis in Geophysics”, The University of Alberta Press, 1975, pp. 106-108.

[3] Wikipedia, “Window function”, https://en.wikipedia.org/wiki/Window_function

[4] W.H. Press, B.P. Flannery, S.A. Teukolsky, and W.T. Vetterling, “Numerical Recipes”, Cambridge University Press, 1986, page 425.

[1] J. F. Kaiser, “Digital Filters” - Ch 7 in “Systems analysis by digital computer”, Editors: F.F. Kuo and J.F. Kaiser, p 218-285. John Wiley and Sons, New York, (1966).

[2] E.R. Kanasewich, “Time Sequence Analysis in Geophysics”, The University of Alberta Press, 1975, pp. 177-178.

[3] Wikipedia, “Window function”, https://en.wikipedia.org/wiki/Window_function

# PYTHON MODULE INDEX

n

`numpy`, 1
`numpy.char`, 575
`numpy.ctypeslib`, 629
`numpy.distutils`, 1595
`numpy.distutils.ccompiler`, 1598
`numpy.distutils.exec_command`, 1600
`numpy.distutils.misc_util`, 1595
`numpy.doc.constants`, 431
`numpy.doc.internals`, 1731
`numpy.dual`, 659
`numpy.fft`, 665
`numpy.lib.format`, 791
`numpy.lib.scimath`, 660
`numpy.linalg`, 793
`numpy.ma`, 237
`numpy.matlib`, 985
`numpy.polynomial.chebyshev`, 1041
`numpy.polynomial.hermite`, 1076
`numpy.polynomial.hermite_e`, 1108
`numpy.polynomial.laguerre`, 1139
`numpy.polynomial.legendre`, 1171
`numpy.polynomial.polynomial`, 1011
`numpy.polynomial.polyutils`, 1202
`numpy.random`, 1224
`numpy.testing`, 1554

# INDEX

## Non-alphabetical

`__abs__()` (*numpy.ma.MaskedArray method*), 307
`__abs__()` (*numpy.ndarray method*), 51
`__add__()` (*numpy.ma.MaskedArray method*), 307
`__add__()` (*numpy.ndarray method*), 51
`__and__()` (*numpy.ma.MaskedArray method*), 308
`__and__()` (*numpy.ndarray method*), 52
`__array__()` (*numpy.class method*), 116
`__array__()` (*numpy.generic method*), 72
`__array__()` (*numpy.ma.MaskedArray method*), 313
`__array__()` (*numpy.ndarray method*), 55
`__array_finalize__()` (*numpy.class method*), 115
`__array_function__()` (*numpy.class method*), 114
`__array_interface__` (*built-in variable*), 419
`__array_interface__` (*numpy.generic attribute*), 61
`__array_prepare__()` (*numpy.class method*), 116
`__array_priority__` (*numpy.class attribute*), 116
`__array_priority__` (*numpy.generic attribute*), 62
`__array_priority__` (*numpy.ma.MaskedArray attribute*), 268
`__array_struct__` (*C variable*), 421
`__array_struct__` (*numpy.generic attribute*), 61
`__array_ufunc__()` (*numpy.class method*), 112
`__array_wrap__()` (*numpy.class method*), 116
`__array_wrap__()` (*numpy.generic method*), 62
`__array_wrap__()` (*numpy.ma.MaskedArray method*), 313
`__array_wrap__()` (*numpy.ndarray method*), 55
`__bool__()` (*numpy.ma.MaskedArray method*), 306
`__bool__()` (*numpy.ndarray method*), 50
`__call__()` (*numpy.errstate method*), 663
`__call__()` (*numpy.poly1d method*), 1211
`__call__()` (*numpy.polynomial.chebyshev.Chebyshev method*), 1042
`__call__()` (*numpy.polynomial.hermite_e.HermiteE method*), 1109
`__call__()` (*numpy.polynomial.hermite.Hermite method*), 1077
`__call__()` (*numpy.polynomial.laguerre.Laguerre method*), 1140
`__call__()` (*numpy.polynomial.legendre.Legendre method*), 1172
`__call__()` (*numpy.polynomial.polynomial.Polynomial method*), 1012
`__call__()` (*numpy.testing.suppress_warnings method*), 1572
`__call__()` (*numpy.vectorize method*), 710
`__complex__()` (*numpy.ndarray method*), 56
`__contains__()` (*numpy.ma.MaskedArray method*), 313
`__contains__()` (*numpy.ndarray method*), 56
`__copy__()` (*numpy.ma.MaskedArray method*), 311
`__copy__()` (*numpy.ndarray method*), 54
`__deepcopy__()` (*numpy.ma.MaskedArray method*), 312
`__deepcopy__()` (*numpy.ndarray method*), 54
`__delitem__()` (*numpy.ma.MaskedArray method*), 313
`__div__()` (*numpy.ma.MaskedArray method*), 307
`__divmod__()` (*numpy.ma.MaskedArray method*), 308
`__divmod__()` (*numpy.ndarray method*), 52
`__eq__()` (*numpy.ma.MaskedArray method*), 305
`__eq__()` (*numpy.ndarray method*), 50
`__float__()` (*numpy.ma.MaskedArray method*), 269
`__float__()` (*numpy.ndarray method*), 56
`__floordiv__()` (*numpy.ma.MaskedArray method*), 307
`__floordiv__()` (*numpy.ndarray method*), 52
`__ge__()` (*numpy.ma.MaskedArray method*), 305
`__ge__()` (*numpy.ndarray method*), 50
`__getitem__()` (*numpy.ma.MaskedArray method*), 313
`__getitem__()` (*numpy.ndarray method*), 56
`__getstate__()` (*numpy.ma.MaskedArray method*), 312
`__gt__()` (*numpy.ma.MaskedArray method*), 305
`__gt__()` (*numpy.ndarray method*), 50
`__iadd__()` (*numpy.ma.MaskedArray method*), 309
`__iadd__()` (*numpy.ndarray method*), 53
`__iand__()` (*numpy.ma.MaskedArray method*), 310
`__iand__()` (*numpy.ndarray method*), 53
`__idiv__()` (*numpy.ma.MaskedArray method*), 309
`__ifloordiv__()` (*numpy.ma.MaskedArray method*), 309

__ifloordiv__() (numpy.ndarray method), 53
__ilshift__() (numpy.ma.MaskedArray method), 310
__ilshift__() (numpy.ndarray method), 53
__imod__() (numpy.ma.MaskedArray method), 309
__imod__() (numpy.ndarray method), 53
__imul__() (numpy.ma.MaskedArray method), 309
__imul__() (numpy.ndarray method), 53
__int__() (numpy.ma.MaskedArray method), 269
__int__() (numpy.ndarray method), 56
__invert__() (numpy.ndarray method), 51
__ior__() (numpy.ma.MaskedArray method), 310
__ior__() (numpy.ndarray method), 53
__ipow__() (numpy.ma.MaskedArray method), 310
__ipow__() (numpy.ndarray method), 53
__irshift__() (numpy.ma.MaskedArray method), 310
__irshift__() (numpy.ndarray method), 53
__isub__() (numpy.ma.MaskedArray method), 309
__isub__() (numpy.ndarray method), 53
__itruediv__() (numpy.ma.MaskedArray method), 309
__itruediv__() (numpy.ndarray method), 53
__ixor__() (numpy.ma.MaskedArray method), 310
__ixor__() (numpy.ndarray method), 54
__le__() (numpy.ma.MaskedArray method), 305
__le__() (numpy.ndarray method), 50
__len__() (numpy.ma.MaskedArray method), 313
__len__() (numpy.ndarray method), 55
__lshift__() (numpy.ma.MaskedArray method), 308
__lshift__() (numpy.ndarray method), 52
__lt__() (numpy.ma.MaskedArray method), 305
__lt__() (numpy.ndarray method), 50
__matmul__() (numpy.ndarray method), 54
__mod__() (numpy.ma.MaskedArray method), 308
__mod__() (numpy.ndarray method), 52
__mul__() (numpy.ma.MaskedArray method), 307
__mul__() (numpy.ndarray method), 51
__ne__() (numpy.ma.MaskedArray method), 306
__ne__() (numpy.ndarray method), 50
__neg__() (numpy.ndarray method), 51
__new__() (numpy.ma.MaskedArray static method), 312
__new__() (numpy.ndarray method), 55
__or__() (numpy.ma.MaskedArray method), 308
__or__() (numpy.ndarray method), 52
__pos__() (numpy.ndarray method), 51
__pow__() (numpy.ma.MaskedArray method), 308
__pow__() (numpy.ndarray method), 52
__radd__() (numpy.ma.MaskedArray method), 307
__rand__() (numpy.ma.MaskedArray method), 308
__rdivmod__() (numpy.ma.MaskedArray method), 308
__reduce__() (numpy.dtype method), 91
__reduce__() (numpy.generic method), 72
__reduce__() (numpy.ma.MaskedArray method), 312
__reduce__() (numpy.ndarray method), 54
__repr__() (numpy.ma.MaskedArray method), 310
__repr__() (numpy.ndarray method), 56
__rfloordiv__() (numpy.ma.MaskedArray method), 307
__rlshift__() (numpy.ma.MaskedArray method), 308
__rmod__() (numpy.ma.MaskedArray method), 308
__rmul__() (numpy.ma.MaskedArray method), 307
__ror__() (numpy.ma.MaskedArray method), 309
__rpow__() (numpy.ma.MaskedArray method), 308
__rrshift__() (numpy.ma.MaskedArray method), 308
__rshift__() (numpy.ma.MaskedArray method), 308
__rshift__() (numpy.ndarray method), 52
__rsub__() (numpy.ma.MaskedArray method), 307
__rtruediv__() (numpy.ma.MaskedArray method), 307
__rxor__() (numpy.ma.MaskedArray method), 309
__setitem__() (numpy.ma.MaskedArray method), 313
__setitem__() (numpy.ndarray method), 56
__setmask__() (numpy.ma.MaskedArray method), 314
__setstate__() (numpy.dtype method), 91
__setstate__() (numpy.generic method), 73
__setstate__() (numpy.ma.MaskedArray method), 312
__setstate__() (numpy.ndarray method), 54
__str__() (numpy.ma.MaskedArray method), 310
__str__() (numpy.ndarray method), 56
__sub__() (numpy.ma.MaskedArray method), 307
__sub__() (numpy.ndarray method), 51
__truediv__() (numpy.ma.MaskedArray method), 307
__truediv__() (numpy.ndarray method), 52
__xor__() (numpy.ma.MaskedArray method), 309
__xor__() (numpy.ndarray method), 52

A

a (C variable), 1687
A() (numpy.matrix property), 118
absolute (in module numpy), 970
abspath() (numpy.DataSource method), 789
accumulate
    ufunc methods, 1729
accumulate() (numpy.ufunc method), 454
add (in module numpy), 943
add() (in module numpy.char), 576
add_data_dir() (numpy.distutils.misc_util.Configuration method), 1603
add_data_files() (numpy.distutils.misc_util.Configuration method), 1602

`add_extension()` (*numpy.distutils.misc_util.Configuration method*), 1605
`add_headers()` (*numpy.distutils.misc_util.Configuration method*), 1604
`add_include_dirs()` (*numpy.distutils.misc_util.Configuration method*), 1604
`add_installed_library()` (*numpy.distutils.misc_util.Configuration method*), 1606
`add_library()` (*numpy.distutils.misc_util.Configuration method*), 1606
`add_npy_pkg_config()` (*numpy.distutils.misc_util.Configuration method*), 1607
`add_scripts()` (*numpy.distutils.misc_util.Configuration method*), 1606
`add_subpackage()` (*numpy.distutils.misc_util.Configuration method*), 1602
`advance()` (*numpy.random.PCG64 method*), 1443
`advance()` (*numpy.random.Philox method*), 1447
`aligned`, 35
`alignment` (*numpy.dtype attribute*), 90
`all` (*in module numpy.ma*), 327
`all()` (*in module numpy*), 846
`all()` (*numpy.generic method*), 64
`all()` (*numpy.ma.MaskedArray method*), 294
`all()` (*numpy.matrix method*), 123
`all()` (*numpy.ndarray method*), 8
`all()` (*numpy.recarray method*), 196
`all()` (*numpy.record method*), 225
`all_strings()` (*in module numpy.distutils.misc_util*), 1596
`allclose()` (*in module numpy*), 863
`allclose()` (*in module numpy.ma*), 410
`allequal()` (*in module numpy.ma*), 409
`allpath()` (*in module numpy.distutils.misc_util*), 1595
`amax()` (*in module numpy*), 1506
`amin()` (*in module numpy*), 1504
`angle()` (*in module numpy*), 960
`anom` (*in module numpy.ma*), 376
`anom()` (*numpy.ma.MaskedArray method*), 295
`anomalies` (*in module numpy.ma*), 377
`any` (*in module numpy.ma*), 327
`any()` (*in module numpy*), 847
`any()` (*numpy.generic method*), 64
`any()` (*numpy.ma.MaskedArray method*), 296
`any()` (*numpy.matrix method*), 124
`any()` (*numpy.ndarray method*), 8
`any()` (*numpy.recarray method*), 196
`any()` (*numpy.record method*), 225
`append()` (*in module numpy*), 553
`append()` (*in module numpy.ma*), 355
`appendpath()` (*in module numpy.distutils.misc_util*), 1595
`apply_along_axis()` (*in module numpy*), 705
`apply_along_axis()` (*in module numpy.ma*), 411
`apply_over_axes()` (*in module numpy*), 707
`apply_over_axes()` (*in module numpy.ma*), 413
`arange` (*in module numpy.ma*), 414
`arange()` (*in module numpy*), 495
`arccos` (*in module numpy*), 878
`arccosh` (*in module numpy*), 894
`arcsin` (*in module numpy*), 877
`arcsinh` (*in module numpy*), 893
`arctan` (*in module numpy*), 880
`arctan2` (*in module numpy*), 883
`arctanh` (*in module numpy*), 895
`argmax` (*in module numpy.ma*), 388
`argmax()` (*in module numpy*), 1495
`argmax()` (*numpy.generic method*), 65
`argmax()` (*numpy.ma.MaskedArray method*), 282
`argmax()` (*numpy.matrix method*), 124
`argmax()` (*numpy.ndarray method*), 9
`argmax()` (*numpy.recarray method*), 196
`argmax()` (*numpy.record method*), 225
`argmin` (*in module numpy.ma*), 389
`argmin()` (*in module numpy*), 1497
`argmin()` (*numpy.generic method*), 65
`argmin()` (*numpy.ma.MaskedArray method*), 283
`argmin()` (*numpy.matrix method*), 125
`argmin()` (*numpy.ndarray method*), 9
`argmin()` (*numpy.recarray method*), 197
`argmin()` (*numpy.record method*), 225
`argpartition()` (*in module numpy*), 1493
`argpartition()` (*numpy.matrix method*), 126
`argpartition()` (*numpy.ndarray method*), 9
`argpartition()` (*numpy.recarray method*), 197
`argsort()` (*in module numpy*), 1488
`argsort()` (*in module numpy.ma*), 392
`argsort()` (*numpy.chararray method*), 170
`argsort()` (*numpy.char.chararray method*), 608
`argsort()` (*numpy.generic method*), 65
`argsort()` (*numpy.ma.MaskedArray method*), 283
`argsort()` (*numpy.matrix method*), 126
`argsort()` (*numpy.ndarray method*), 9
`argsort()` (*numpy.recarray method*), 197
`argsort()` (*numpy.record method*), 225
`argwhere()` (*in module numpy*), 1499
`arithmetic`, 50, 305
`around` (*in module numpy.ma*), 406
`around()` (*in module numpy*), 896
`array`
    `C-API`, 1647
    `interface`, 419
    `protocol`, 419
`array iterator`, 233, 1724

array scalars, 1725
array() (*in module numpy*), 476
array() (*in module numpy.char*), 601
array() (*in module numpy.core.defchararray*), 190
array() (*in module numpy.core.records*), 491
array() (*in module numpy.ma*), 239
array2string() (*in module numpy*), 775
array_equal() (*in module numpy*), 866
array_equiv() (*in module numpy*), 867
array_repr() (*in module numpy*), 778
array_split() (*in module numpy*), 545
array_str() (*in module numpy*), 779
Arrayterator (*class in numpy.lib*), 760
as_array() (*in module numpy.ctypeslib*), 629
as_ctypes() (*in module numpy.ctypeslib*), 629
as_ctypes_type() (*in module numpy.ctypeslib*), 629
as_list() (*in module numpy.distutils.misc_util*), 1596
as_series() (*in module numpy.polynomial.polyutils*), 1203
as_strided() (*in module numpy.lib.stride_tricks*), 743
asanyarray() (*in module numpy*), 480
asanyarray() (*in module numpy.ma*), 242
asarray() (*in module numpy*), 479
asarray() (*in module numpy.char*), 602
asarray() (*in module numpy.core.defchararray*), 494
asarray() (*in module numpy.ma*), 241
asarray_chkfinite() (*in module numpy*), 530
ascontiguousarray() (*in module numpy*), 481
asfarray() (*in module numpy*), 528
asfortranarray() (*in module numpy*), 529
asmatrix() (*in module numpy*), 157
asscalar() (*in module numpy*), 531
assert_allclose() (*in module numpy.testing*), 1558
assert_almost_equal() (*in module numpy.testing*), 1555
assert_approx_equal() (*in module numpy.testing*), 1556
assert_array_almost_equal() (*in module numpy.testing*), 1557
assert_array_almost_equal_nulp() (*in module numpy.testing*), 1559
assert_array_equal() (*in module numpy.testing*), 1561
assert_array_less() (*in module numpy.testing*), 1562
assert_array_max_ulp() (*in module numpy.testing*), 1560
assert_equal() (*in module numpy.testing*), 1564
assert_raises() (*in module numpy.testing*), 1565
assert_raises_regex() (*in module numpy.testing*), 1565
assert_string_equal() (*in module numpy.testing*), 1566
assert_warns() (*in module numpy.testing*), 1565
astype() (*numpy.chararray method*), 168
astype() (*numpy.char.chararray method*), 607
astype() (*numpy.generic method*), 65
astype() (*numpy.ma.MaskedArray method*), 270
astype() (*numpy.matrix method*), 126
astype() (*numpy.ndarray method*), 10
astype() (*numpy.recarray method*), 197
astype() (*numpy.record method*), 225
at() (*numpy.ufunc method*), 458
atleast_1d (*in module numpy.ma*), 341
atleast_1d() (*in module numpy*), 522
atleast_2d (*in module numpy.ma*), 341
atleast_2d() (*in module numpy*), 522
atleast_3d (*in module numpy.ma*), 342
atleast_3d() (*in module numpy*), 523
attributes
    ufunc, 448
average() (*in module numpy*), 1524
average() (*in module numpy.ma*), 378
axis, 48

## B

b (*C variable*), 1687
bartlett() (*in module numpy*), 1579
base, 3
base (*numpy.dtype attribute*), 90
base (*numpy.generic attribute*), 61
base (*numpy.ma.MaskedArray attribute*), 260
base (*numpy.ndarray attribute*), 41
base_repr() (*in module numpy*), 787
baseclass (*numpy.ma.MaskedArray attribute*), 260
basis() (*numpy.polynomial.chebyshev.Chebyshev class method*), 1042
basis() (*numpy.polynomial.hermite_e.HermiteE class method*), 1109
basis() (*numpy.polynomial.hermite.Hermite class method*), 1077
basis() (*numpy.polynomial.laguerre.Laguerre class method*), 1140
basis() (*numpy.polynomial.legendre.Legendre class method*), 1172
basis() (*numpy.polynomial.polynomial.Polynomial class method*), 1012
beta() (*in module numpy.random*), 1368
beta() (*numpy.random.Generator method*), 1236
beta() (*numpy.random.RandomState method*), 1307
binary_repr() (*in module numpy*), 574
bincount() (*in module numpy*), 1548
binomial() (*in module numpy.random*), 1369
binomial() (*numpy.random.Generator method*), 1236
binomial() (*numpy.random.RandomState method*), 1308
binomial_t (*C type*), 1464

bit_generator (*numpy.random.Generator attribute*), 1228
bitgen_t (*C type*), 1462
BitGenerator (*class in numpy.random*), 1437
bitwise_and (*in module numpy*), 564
bitwise_or (*in module numpy*), 565
bitwise_xor (*in module numpy*), 566
blackman() (*in module numpy*), 1581
block() (*in module numpy*), 536
blue_text() (*in module numpy.distutils.misc_util*), 1596
bmat() (*in module numpy*), 158
broadcast (*class in numpy*), 235
broadcast_arrays() (*in module numpy*), 525
broadcast_to() (*in module numpy*), 524
broadcastable, 439
broadcasting, 439, 1724
buffers, 440
busday_count() (*in module numpy*), 638
busday_offset() (*in module numpy*), 636
busdaycalendar (*class in numpy*), 634
byte_bounds() (*in module numpy*), 994
byteorder (*numpy.dtype attribute*), 86
bytes() (*in module numpy.random*), 1370
bytes() (*numpy.random.Generator method*), 1232
bytes() (*numpy.random.RandomState method*), 1303
byteswap() (*numpy.generic method*), 65
byteswap() (*numpy.ma.MaskedArray method*), 271
byteswap() (*numpy.matrix method*), 128
byteswap() (*numpy.ndarray method*), 11
byteswap() (*numpy.recarray method*), 199
byteswap() (*numpy.record method*), 226

C

c_ (*in module numpy*), 716
can_cast() (*in module numpy*), 640
C-API
    array, 1647
    iterator, 1689, 1705
    ndarray, 1647, 1689
    ufunc, 1706, 1712
capitalize() (*in module numpy.char*), 577
cast() (*numpy.polynomial.chebyshev.Chebyshev class method*), 1043
cast() (*numpy.polynomial.hermite_e.HermiteE class method*), 1109
cast() (*numpy.polynomial.hermite.Hermite class method*), 1078
cast() (*numpy.polynomial.laguerre.Laguerre class method*), 1141
cast() (*numpy.polynomial.legendre.Legendre class method*), 1172
cast() (*numpy.polynomial.polynomial.Polynomial class method*), 1012
casting rules
    ufunc, 444
cbrt (*in module numpy*), 968
ceil (*in module numpy*), 900
center() (*in module numpy.char*), 578
cffi (*numpy.random.MT19937 attribute*), 1441
cffi (*numpy.random.PCG64 attribute*), 1444
cffi (*numpy.random.Philox attribute*), 1448
cffi (*numpy.random.SFC64 attribute*), 1450
char (*numpy.dtype attribute*), 84
character arrays, 163
chararray (*class in numpy*), 164
chararray (*class in numpy.char*), 602
cheb2poly() (*in module numpy.polynomial.chebyshev*), 1073
chebadd() (*in module numpy.polynomial.chebyshev*), 1052
chebcompanion() (*in module numpy.polynomial.chebyshev*), 1068
chebder() (*in module numpy.polynomial.chebyshev*), 1060
chebdiv() (*in module numpy.polynomial.chebyshev*), 1055
chebdomain (*in module numpy.polynomial.chebyshev*), 1052
chebfit() (*in module numpy.polynomial.chebyshev*), 1069
chebfromroots() (*in module numpy.polynomial.chebyshev*), 1063
chebgauss() (*in module numpy.polynomial.chebyshev*), 1067
chebgrid2d() (*in module numpy.polynomial.chebyshev*), 1059
chebgrid3d() (*in module numpy.polynomial.chebyshev*), 1059
chebint() (*in module numpy.polynomial.chebyshev*), 1061
chebinterpolate() (*in module numpy.polynomial.chebyshev*), 1074
chebline() (*in module numpy.polynomial.chebyshev*), 1072
chebmul() (*in module numpy.polynomial.chebyshev*), 1054
chebmulx() (*in module numpy.polynomial.chebyshev*), 1053
chebone (*in module numpy.polynomial.chebyshev*), 1052
chebpow() (*in module numpy.polynomial.chebyshev*), 1055
chebpts1() (*in module numpy.polynomial.chebyshev*), 1071
chebpts2() (*in module numpy.polynomial.chebyshev*), 1071
chebroots() (*in module numpy.polynomial.chebyshev*), 1064

chebsub() (*in module numpy.polynomial.chebyshev*), 1053
chebtrim() (*in module numpy.polynomial.chebyshev*), 1072
chebval() (*in module numpy.polynomial.chebyshev*), 1056
chebval2d() (*in module numpy.polynomial.chebyshev*), 1057
chebval3d() (*in module numpy.polynomial.chebyshev*), 1058
chebvander() (*in module numpy.polynomial.chebyshev*), 1065
chebvander2d() (*in module numpy.polynomial.chebyshev*), 1065
chebvander3d() (*in module numpy.polynomial.chebyshev*), 1066
chebweight() (*in module numpy.polynomial.chebyshev*), 1068
chebx (*in module numpy.polynomial.chebyshev*), 1052
Chebyshev (*class in numpy.polynomial.chebyshev*), 1041
chebzero (*in module numpy.polynomial.chebyshev*), 1052
chisquare() (*in module numpy.random*), 1371
chisquare() (*numpy.random.Generator method*), 1237
chisquare() (*numpy.random.RandomState method*), 1309
choice() (*in module numpy.random*), 1372
choice() (*numpy.random.Generator method*), 1231
choice() (*numpy.random.RandomState method*), 1302
cholesky() (*in module numpy.linalg*), 815
choose() (*in module numpy*), 737
choose() (*in module numpy.ma*), 415
choose() (*numpy.generic method*), 65
choose() (*numpy.ma.MaskedArray method*), 285
choose() (*numpy.matrix method*), 129
choose() (*numpy.ndarray method*), 12
choose() (*numpy.recarray method*), 200
choose() (*numpy.record method*), 226
clip() (*in module numpy*), 966
clip() (*in module numpy.ma*), 407
clip() (*numpy.generic method*), 66
clip() (*numpy.ma.MaskedArray method*), 296
clip() (*numpy.matrix method*), 129
clip() (*numpy.ndarray method*), 12
clip() (*numpy.recarray method*), 200
clip() (*numpy.record method*), 226
close() (*numpy.nditer method*), 756
clump_masked() (*in module numpy.ma*), 364
clump_unmasked() (*in module numpy.ma*), 364
code generation, 1617
column_stack (*in module numpy.ma*), 347
column_stack() (*in module numpy*), 542
column-major, 35
common_fill_value() (*in module numpy.ma*), 372
common_type() (*in module numpy*), 645
compare_chararrays() (*in module numpy.char*), 592
comparison, 50, 305
compress() (*in module numpy*), 739
compress() (*numpy.generic method*), 66
compress() (*numpy.ma.MaskedArray method*), 285
compress() (*numpy.matrix method*), 129
compress() (*numpy.ndarray method*), 12
compress() (*numpy.recarray method*), 200
compress() (*numpy.record method*), 226
compress_cols() (*in module numpy.ma*), 369
compress_rowcols() (*in module numpy.ma*), 369
compress_rows() (*in module numpy.ma*), 370
compressed() (*in module numpy.ma*), 370
compressed() (*numpy.ma.MaskedArray method*), 272
concatenate() (*in module numpy*), 533
concatenate() (*in module numpy.ma*), 347
cond() (*in module numpy.linalg*), 831
Configuration (*class in numpy.distutils.misc_util*), 1601
conj (*in module numpy*), 962
conj() (*numpy.ma.MaskedArray method*), 296
conj() (*numpy.matrix method*), 129
conj() (*numpy.ndarray method*), 12
conj() (*numpy.recarray method*), 200
conjugate (*in module numpy*), 963
conjugate (*in module numpy.ma*), 379
conjugate() (*numpy.generic method*), 66
conjugate() (*numpy.ma.MaskedArray method*), 296
conjugate() (*numpy.matrix method*), 129
conjugate() (*numpy.ndarray method*), 13
conjugate() (*numpy.recarray method*), 200
conjugate() (*numpy.record method*), 226
construction
    from dict, dtype, 82
    from dtype, dtype, 78
    from list, dtype, 81
    from None, dtype, 78
    from string, dtype, 79
    from tuple, dtype, 81
    from type, dtype, 78
container (*class in numpy.lib.user_array*), 233
container class, 233
contiguous, 35
convert() (*numpy.polynomial.chebyshev.Chebyshev method*), 1043
convert() (*numpy.polynomial.hermite_e.HermiteE method*), 1110
convert() (*numpy.polynomial.hermite.Hermite method*), 1078
convert() (*numpy.polynomial.laguerre.Laguerre method*), 1141

`convert()` (*numpy.polynomial.legendre.Legendre method*), 1173
`convert()` (*numpy.polynomial.polynomial.Polynomial method*), 1013
`convolve()` (*in module numpy*), 964
`copy` (*in module numpy.ma*), 317
`copy()` (*in module numpy*), 482
`copy()` (*numpy.chararray method*), 170
`copy()` (*numpy.char.chararray method*), 609
`copy()` (*numpy.flatiter method*), 759
`copy()` (*numpy.generic method*), 66
`copy()` (*numpy.ma.MaskedArray method*), 293
`copy()` (*numpy.matrix method*), 130
`copy()` (*numpy.ndarray method*), 13
`copy()` (*numpy.nditer method*), 756
`copy()` (*numpy.polynomial.chebyshev.Chebyshev method*), 1044
`copy()` (*numpy.polynomial.hermite_e.HermiteE method*), 1111
`copy()` (*numpy.polynomial.hermite.Hermite method*), 1079
`copy()` (*numpy.polynomial.laguerre.Laguerre method*), 1142
`copy()` (*numpy.polynomial.legendre.Legendre method*), 1174
`copy()` (*numpy.polynomial.polynomial.Polynomial method*), 1014
`copy()` (*numpy.recarray method*), 201
`copy()` (*numpy.record method*), 226
`copysign` (*in module numpy*), 936
`copyto()` (*in module numpy*), 513
`C-order`, 35
`corrcoef()` (*in module numpy*), 1537
`corrcoef()` (*in module numpy.ma*), 380
`correlate()` (*in module numpy*), 1538
`cos` (*in module numpy*), 874
`cosh` (*in module numpy*), 890
`count` (*in module numpy.ma*), 327
`count()` (*in module numpy.char*), 593
`count()` (*numpy.chararray method*), 171
`count()` (*numpy.char.chararray method*), 609
`count()` (*numpy.ma.MaskedArray method*), 316
`count_masked()` (*in module numpy.ma*), 328
`count_nonzero()` (*in module numpy*), 1503
`cov()` (*in module numpy*), 1539
`cov()` (*in module numpy.ma*), 381
`cpu` (*in module numpy.distutils.cpuinfo*), 1599
`cross()` (*in module numpy*), 919
`ctypes` (*numpy.ma.MaskedArray attribute*), 261
`ctypes` (*numpy.ndarray attribute*), 44
`ctypes` (*numpy.random.MT19937 attribute*), 1441
`ctypes` (*numpy.random.PCG64 attribute*), 1445
`ctypes` (*numpy.random.Philox attribute*), 1449
`ctypes` (*numpy.random.SFC64 attribute*), 1451
`ctypes_load_library()` (*in module numpy.ctypeslib*), 630
`cumprod` (*in module numpy.ma*), 382
`cumprod()` (*in module numpy*), 909
`cumprod()` (*numpy.generic method*), 66
`cumprod()` (*numpy.ma.MaskedArray method*), 297
`cumprod()` (*numpy.matrix method*), 130
`cumprod()` (*numpy.ndarray method*), 13
`cumprod()` (*numpy.recarray method*), 201
`cumprod()` (*numpy.record method*), 226
`cumsum` (*in module numpy.ma*), 382
`cumsum()` (*in module numpy*), 910
`cumsum()` (*numpy.generic method*), 66
`cumsum()` (*numpy.ma.MaskedArray method*), 297
`cumsum()` (*numpy.matrix method*), 131
`cumsum()` (*numpy.ndarray method*), 14
`cumsum()` (*numpy.recarray method*), 202
`cumsum()` (*numpy.record method*), 227
`cutdeg()` (*numpy.polynomial.chebyshev.Chebyshev method*), 1044
`cutdeg()` (*numpy.polynomial.hermite_e.HermiteE method*), 1111
`cutdeg()` (*numpy.polynomial.hermite.Hermite method*), 1079
`cutdeg()` (*numpy.polynomial.laguerre.Laguerre method*), 1142
`cutdeg()` (*numpy.polynomial.legendre.Legendre method*), 1174
`cutdeg()` (*numpy.polynomial.polynomial.Polynomial method*), 1014
`cyan_text()` (*in module numpy.distutils.misc_util*), 1596
`cyg2win32()` (*in module numpy.distutils.misc_util*), 1596

## D

`data` (*numpy.generic attribute*), 61
`data` (*numpy.ma.MaskedArray attribute*), 259
`data` (*numpy.ndarray attribute*), 40
`DataSource` (*class in numpy*), 788
`datetime_as_string()` (*in module numpy*), 632
`datetime_data()` (*in module numpy*), 633
`debug_print()` (*numpy.nditer method*), 756
`decode()` (*in module numpy.char*), 578
`decode()` (*numpy.chararray method*), 171
`decode()` (*numpy.char.chararray method*), 609
`decorate_methods()` (*in module numpy.testing*), 1569
`default_config_dict()` (*in module numpy.distutils.misc_util*), 1595
`default_fill_value()` (*in module numpy.ma*), 372
`default_rng()` (*in module numpy.random*), 1227
`deg2rad` (*in module numpy*), 887

degree() (*numpy.polynomial.chebyshev.Chebyshev method*), 1044
degree() (*numpy.polynomial.hermite_e.HermiteE method*), 1111
degree() (*numpy.polynomial.hermite.Hermite method*), 1079
degree() (*numpy.polynomial.laguerre.Laguerre method*), 1142
degree() (*numpy.polynomial.legendre.Legendre method*), 1174
degree() (*numpy.polynomial.polynomial.Polynomial method*), 1014
degrees (*in module numpy*), 884
delete() (*in module numpy*), 550
deprecate() (*in module numpy*), 998
deprecate_with_doc() (*in module numpy*), 999
deprecated() (*in module numpy.testing.dec*), 1567
deriv() (*numpy.poly1d method*), 1211
deriv() (*numpy.polynomial.chebyshev.Chebyshev method*), 1045
deriv() (*numpy.polynomial.hermite_e.HermiteE method*), 1111
deriv() (*numpy.polynomial.hermite.Hermite method*), 1080
deriv() (*numpy.polynomial.laguerre.Laguerre method*), 1143
deriv() (*numpy.polynomial.legendre.Legendre method*), 1174
deriv() (*numpy.polynomial.polynomial.Polynomial method*), 1014
descr (*numpy.dtype attribute*), 90
det() (*in module numpy.linalg*), 832
diag() (*in module numpy*), 507
diag() (*in module numpy.ma*), 394
diag_indices() (*in module numpy*), 726
diag_indices_from() (*in module numpy*), 727
diagflat() (*in module numpy*), 508
diagonal() (*in module numpy*), 740
diagonal() (*numpy.generic method*), 66
diagonal() (*numpy.ma.MaskedArray method*), 286
diagonal() (*numpy.matrix method*), 131
diagonal() (*numpy.ndarray method*), 14
diagonal() (*numpy.recarray method*), 202
diagonal() (*numpy.record method*), 227
dict_append() (*in module numpy.distutils.misc_util*), 1595
diff() (*in module numpy*), 914
digitize() (*in module numpy*), 1552
dirichlet() (*in module numpy.random*), 1374
dirichlet() (*numpy.random.Generator method*), 1239
dirichlet() (*numpy.random.RandomState method*), 1311
disp() (*in module numpy*), 1000
distutils, 1595
divide (*in module numpy*), 947
divmod (*in module numpy*), 959
dot() (*in module numpy*), 794
dot() (*in module numpy.ma*), 394
dot() (*numpy.matrix method*), 131
dot() (*numpy.ndarray method*), 14
dot() (*numpy.recarray method*), 202
dot_join() (*in module numpy.distutils.misc_util*), 1596
dsplit() (*in module numpy*), 545
dstack (*in module numpy.ma*), 348
dstack() (*in module numpy*), 541
dtype, 1724
    construction from dict, 82
    construction from dtype, 78
    construction from list, 81
    construction from None, 78
    construction from string, 79
    construction from tuple, 81
    construction from type, 78
    field, 73
    scalar, 73
    sub-array, 73, 81
dtype (*class in numpy*), 74
dtype (*numpy.generic attribute*), 61
dtype (*numpy.ndarray attribute*), 42
dtype() (*numpy.ma.MaskedArray property*), 262
dump() (*numpy.chararray method*), 171
dump() (*numpy.char.chararray method*), 610
dump() (*numpy.generic method*), 67
dump() (*numpy.ma.MaskedArray method*), 293
dump() (*numpy.matrix method*), 131
dump() (*numpy.ndarray method*), 15
dump() (*numpy.recarray method*), 202
dump() (*numpy.record method*), 227
dumps() (*numpy.chararray method*), 171
dumps() (*numpy.char.chararray method*), 610
dumps() (*numpy.generic method*), 67
dumps() (*numpy.ma.MaskedArray method*), 294
dumps() (*numpy.matrix method*), 132
dumps() (*numpy.ndarray method*), 15
dumps() (*numpy.recarray method*), 203
dumps() (*numpy.record method*), 227

# E

e (*in module numpy*), 434
ediff1d() (*in module numpy*), 915
ediff1d() (*in module numpy.ma*), 416
eig() (*in module numpy.linalg*), 821
eigh() (*in module numpy.linalg*), 823
eigvals() (*in module numpy.linalg*), 825
eigvalsh() (*in module numpy.linalg*), 827
einsum() (*in module numpy*), 805
einsum_path() (*in module numpy*), 811
ellipsis, 92

empty (*in module numpy.ma*), 320
empty() (*in module numpy*), 465
empty() (*in module numpy.matlib*), 985
empty_like (*in module numpy.ma*), 321
empty_like() (*in module numpy*), 466
enable_external_loop() (*numpy.nditer method*), 756
encode() (*in module numpy.char*), 579
encode() (*numpy.chararray method*), 171
encode() (*numpy.char.chararray method*), 610
endswith() (*in module numpy.char*), 594
endswith() (*numpy.chararray method*), 171
endswith() (*numpy.char.chararray method*), 610
equal (*in module numpy*), 871
equal() (*in module numpy.char*), 589
error handling, 441
errstate (*class in numpy*), 662
euler_gamma (*in module numpy*), 434
exists() (*numpy.DataSource method*), 789
exp (*in module numpy*), 922
exp2 (*in module numpy*), 925
expand_dims() (*in module numpy*), 525
expand_dims() (*in module numpy.ma*), 343
expandtabs() (*in module numpy.char*), 580
expandtabs() (*numpy.chararray method*), 171
expandtabs() (*numpy.char.chararray method*), 610
expm1 (*in module numpy*), 924
exponential() (*in module numpy.random*), 1375
exponential() (*numpy.random.Generator method*), 1240
exponential() (*numpy.random.RandomState method*), 1312
Extension (*class in numpy.distutils.core*), 1599
extract() (*in module numpy*), 1502
eye() (*in module numpy*), 468
eye() (*in module numpy.matlib*), 988

# F

f() (*in module numpy.random*), 1376
f() (*numpy.random.Generator method*), 1241
f() (*numpy.random.RandomState method*), 1313
fabs (*in module numpy*), 971
fft() (*in module numpy.fft*), 665
fft2() (*in module numpy.fft*), 669
fftfreq() (*in module numpy.fft*), 688
fftn() (*in module numpy.fft*), 673
fftshift() (*in module numpy.fft*), 689
field
    dtype, 73
fields (*numpy.dtype attribute*), 87
fill() (*numpy.chararray method*), 172
fill() (*numpy.char.chararray method*), 610
fill() (*numpy.generic method*), 67
fill() (*numpy.ma.MaskedArray method*), 286
fill() (*numpy.matrix method*), 132
fill() (*numpy.ndarray method*), 15
fill() (*numpy.recarray method*), 203
fill() (*numpy.record method*), 227
fill_diagonal() (*in module numpy*), 748
fill_value (*numpy.ma.MaskedArray attribute*), 259
filled() (*in module numpy.ma*), 371
filled() (*numpy.ma.MaskedArray method*), 272
filter() (*numpy.testing.suppress_warnings method*), 1572
filter_sources() (*in module numpy.distutils.misc_util*), 1596
find() (*in module numpy.char*), 595
find() (*numpy.chararray method*), 172
find() (*numpy.char.chararray method*), 611
find_common_type() (*in module numpy*), 654
finfo (*class in numpy*), 647
fit() (*numpy.polynomial.chebyshev.Chebyshev class method*), 1045
fit() (*numpy.polynomial.hermite_e.HermiteE class method*), 1112
fit() (*numpy.polynomial.hermite.Hermite class method*), 1080
fit() (*numpy.polynomial.laguerre.Laguerre class method*), 1143
fit() (*numpy.polynomial.legendre.Legendre class method*), 1175
fit() (*numpy.polynomial.polynomial.Polynomial class method*), 1015
fix() (*in module numpy*), 899
fix_invalid() (*in module numpy.ma*), 243
flags (*numpy.dtype attribute*), 89
flags (*numpy.generic attribute*), 60
flags (*numpy.ma.MaskedArray attribute*), 263
flags (*numpy.ndarray attribute*), 36
flat (*numpy.generic attribute*), 61
flat (*numpy.ndarray attribute*), 43
flat() (*numpy.ma.MaskedArray property*), 268
flatiter (*class in numpy*), 758
flatnonzero() (*in module numpy*), 1500
flatnotmasked_contiguous() (*in module numpy.ma*), 360
flatnotmasked_edges() (*in module numpy.ma*), 361
flatten() (*numpy.chararray method*), 172
flatten() (*numpy.char.chararray method*), 611
flatten() (*numpy.generic method*), 67
flatten() (*numpy.ma.MaskedArray method*), 277
flatten() (*numpy.matrix method*), 132
flatten() (*numpy.ndarray method*), 16
flatten() (*numpy.recarray method*), 203
flatten() (*numpy.record method*), 227
flexible, 60
flip() (*in module numpy*), 558

fliplr() (*in module numpy*), 559
flipud() (*in module numpy*), 560
float_power (*in module numpy*), 952
floor (*in module numpy*), 899
floor_divide (*in module numpy*), 951
flush() (*numpy.memmap method*), 163
fmax (*in module numpy*), 977
fmin (*in module numpy*), 979
fmod (*in module numpy*), 953
format_float_positional() (*in module numpy*), 779
format_float_scientific() (*in module numpy*), 781
format_parser (*class in numpy*), 646
Fortran-order, 35
frexp (*in module numpy*), 937
from dict
    dtype construction, 82
from dtype
    dtype construction, 78
from list
    dtype construction, 81
from None
    dtype construction, 78
from string
    dtype construction, 79
from tuple
    dtype construction, 81
from type
    dtype construction, 78
fromarrays() (*in module numpy.core.records*), 491
frombuffer (*in module numpy.ma*), 318
frombuffer() (*in module numpy*), 483
fromfile() (*in module numpy*), 484
fromfile() (*in module numpy.core.records*), 493
fromfunction (*in module numpy.ma*), 318
fromfunction() (*in module numpy*), 485
fromiter() (*in module numpy*), 486
frompyfunc() (*in module numpy*), 710
fromrecords() (*in module numpy.core.records*), 492
fromregex() (*in module numpy*), 773
fromroots() (*numpy.polynomial.chebyshev.Chebyshev class method*), 1046
fromroots() (*numpy.polynomial.hermite_e.HermiteE class method*), 1113
fromroots() (*numpy.polynomial.hermite.Hermite class method*), 1081
fromroots() (*numpy.polynomial.laguerre.Laguerre class method*), 1144
fromroots() (*numpy.polynomial.legendre.Legendre class method*), 1176
fromroots() (*numpy.polynomial.polynomial.Polynomial class method*), 1016
fromstring() (*in module numpy*), 487
fromstring() (*in module numpy.core.records*), 492
full() (*in module numpy*), 474
full_like() (*in module numpy*), 475
fv() (*in module numpy*), 693

G

gamma() (*in module numpy.random*), 1377
gamma() (*numpy.random.Generator method*), 1242
gamma() (*numpy.random.RandomState method*), 1315
gcd (*in module numpy*), 942
generate_config_py() (*in module numpy.distutils.misc_util*), 1595
generate_state() (*numpy.random.SeedSequence method*), 1453
Generator (*class in numpy.random*), 1227
generic (*class in numpy*), 62
genfromtxt() (*in module numpy*), 770
geometric() (*in module numpy.random*), 1379
geometric() (*numpy.random.Generator method*), 1243
geometric() (*numpy.random.RandomState method*), 1316
geomspace() (*in module numpy*), 500
get_build_architecture() (*in module numpy.distutils.misc_util*), 1596
get_build_temp_dir() (*numpy.distutils.misc_util.Configuration method*), 1609
get_cmd() (*in module numpy.distutils.misc_util*), 1595
get_config_cmd() (*numpy.distutils.misc_util.Configuration method*), 1609
get_data_files() (*in module numpy.distutils.misc_util*), 1596
get_dependencies() (*in module numpy.distutils.misc_util*), 1596
get_distribution() (*numpy.distutils.misc_util.Configuration method*), 1601
get_ext_source_files() (*in module numpy.distutils.misc_util*), 1596
get_fill_value() (*numpy.ma.MaskedArray method*), 315
get_frame() (*in module numpy.distutils.misc_util*), 1596
get_include() (*in module numpy*), 997
get_info() (*in module numpy.distutils.misc_util*), 1596
get_info() (*in module numpy.distutils.system_info*), 1601
get_info() (*numpy.distutils.misc_util.Configuration method*), 1610
get_language() (*in module numpy.distutils.misc_util*), 1596
get_lib_source_files() (*in module numpy.distutils.misc_util*), 1596

get_mathlibs() (*in module numpy.distutils.misc_util*), 1595
get_num_build_jobs() (*in module numpy.distutils.misc_util*), 1598
get_numpy_include_dirs() (*in module numpy.distutils.misc_util*), 1595
get_pkg_info() (*in module numpy.distutils.misc_util*), 1597
get_printoptions() (*in module numpy*), 785
get_script_files() (*in module numpy.distutils.misc_util*), 1596
get_standard_file() (*in module numpy.distutils.system_info*), 1601
get_state() (*in module numpy.random*), 1380
get_state() (*numpy.random.RandomState method*), 1294
get_subpackage() (*numpy.distutils.misc_util.Configuration method*), 1601
get_version() (*numpy.distutils.misc_util.Configuration method*), 1609
getA() (*numpy.matrix method*), 133
getA1() (*numpy.matrix method*), 133
getbufsize() (*in module numpy*), 992
getdata() (*in module numpy.ma*), 331
getdomain() (*in module numpy.polynomial.polyutils*), 1205
geterr() (*in module numpy*), 660
geterrcall() (*in module numpy*), 661
geterrobj() (*in module numpy*), 664
getfield() (*numpy.chararray method*), 173
getfield() (*numpy.char.chararray method*), 612
getfield() (*numpy.generic method*), 67
getfield() (*numpy.matrix method*), 136
getfield() (*numpy.ndarray method*), 16
getfield() (*numpy.recarray method*), 204
getfield() (*numpy.record method*), 227
getH() (*numpy.matrix method*), 134
getI() (*numpy.matrix method*), 134
getitem
    ndarray special methods, 92
getmask() (*in module numpy.ma*), 329
getmaskarray() (*in module numpy.ma*), 330
getT() (*numpy.matrix method*), 135
gpaths() (*in module numpy.distutils.misc_util*), 1596
gradient() (*in module numpy*), 916
greater (*in module numpy*), 868
greater() (*in module numpy.char*), 591
greater_equal (*in module numpy*), 869
greater_equal() (*in module numpy.char*), 590
green_text() (*in module numpy.distutils.misc_util*), 1595
gumbel() (*in module numpy.random*), 1381
gumbel() (*numpy.random.Generator method*), 1244
gumbel() (*numpy.random.RandomState method*), 1318

H

H() (*numpy.matrix property*), 117
hamming() (*in module numpy*), 1583
hanning() (*in module numpy*), 1586
harden_mask (*in module numpy.ma*), 368
harden_mask() (*numpy.ma.MaskedArray method*), 314
hardmask (*numpy.ma.MaskedArray attribute*), 260
has_cxx_sources() (*in module numpy.distutils.misc_util*), 1596
has_f_sources() (*in module numpy.distutils.misc_util*), 1596
has_samecoef() (*numpy.polynomial.chebyshev.Chebyshev method*), 1047
has_samecoef() (*numpy.polynomial.hermite_e.HermiteE method*), 1113
has_samecoef() (*numpy.polynomial.hermite.Hermite method*), 1082
has_samecoef() (*numpy.polynomial.laguerre.Laguerre method*), 1145
has_samecoef() (*numpy.polynomial.legendre.Legendre method*), 1176
has_samecoef() (*numpy.polynomial.polynomial.Polynomial method*), 1016
has_samedomain() (*numpy.polynomial.chebyshev.Chebyshev method*), 1047
has_samedomain() (*numpy.polynomial.hermite_e.HermiteE method*), 1114
has_samedomain() (*numpy.polynomial.hermite.Hermite method*), 1082
has_samedomain() (*numpy.polynomial.laguerre.Laguerre method*), 1145
has_samedomain() (*numpy.polynomial.legendre.Legendre method*), 1177
has_samedomain() (*numpy.polynomial.polynomial.Polynomial method*), 1017
has_sametype() (*numpy.polynomial.chebyshev.Chebyshev method*), 1047
has_sametype() (*numpy.polynomial.hermite_e.HermiteE method*), 1114
has_sametype() (*numpy.polynomial.hermite.Hermite method*), 1082
has_sametype() (*numpy.polynomial.laguerre.Laguerre method*), 1145
has_sametype() (*numpy.polynomial.legendre.Legendre method*), 1177
has_sametype() (*numpy.polynomial.polynomial.Polynomial method*), 1017
has_samewindow() (*numpy.polynomial.chebyshev.Chebyshev method*), 1048
has_samewindow() (*numpy.polynomial.hermite_e.HermiteE method*), 1114
has_samewindow() (*numpy.polynomial.hermite.Hermite method*), 1083

has_samewindow() (*numpy.polynomial.laguerre.Laguerre method*), 1146
has_samewindow() (*numpy.polynomial.legendre.Legendre method*), 1177
has_samewindow() (*numpy.polynomial.polynomial.Polynomial method*), 1017
hasobject (*numpy.dtype attribute*), 89
have_f77c() (*numpy.distutils.misc_util.Configuration method*), 1609
have_f90c() (*numpy.distutils.misc_util.Configuration method*), 1609
heaviside (*in module numpy*), 973
herm2poly() (*in module numpy.polynomial.hermite*), 1106
hermadd() (*in module numpy.polynomial.hermite*), 1086
hermcompanion() (*in module numpy.polynomial.hermite*), 1102
hermder() (*in module numpy.polynomial.hermite*), 1095
hermdiv() (*in module numpy.polynomial.hermite*), 1089
hermdomain (*in module numpy.polynomial.hermite*), 1086
herme2poly() (*in module numpy.polynomial.hermite_e*), 1138
hermeadd() (*in module numpy.polynomial.hermite_e*), 1118
hermecompanion() (*in module numpy.polynomial.hermite_e*), 1134
hermeder() (*in module numpy.polynomial.hermite_e*), 1126
hermediv() (*in module numpy.polynomial.hermite_e*), 1121
hermedomain (*in module numpy.polynomial.hermite_e*), 1118
hermefit() (*in module numpy.polynomial.hermite_e*), 1134
hermefromroots() (*in module numpy.polynomial.hermite_e*), 1129
hermegauss() (*in module numpy.polynomial.hermite_e*), 1133
hermegrid2d() (*in module numpy.polynomial.hermite_e*), 1125
hermegrid3d() (*in module numpy.polynomial.hermite_e*), 1125
hermeint() (*in module numpy.polynomial.hermite_e*), 1127
hermeline() (*in module numpy.polynomial.hermite_e*), 1137
hermemul() (*in module numpy.polynomial.hermite_e*), 1120
hermemulx() (*in module numpy.polynomial.hermite_e*), 1119
hermeone (*in module numpy.polynomial.hermite_e*), 1118
hermepow() (*in module numpy.polynomial.hermite_e*), 1121
hermeroots() (*in module numpy.polynomial.hermite_e*), 1130
hermesub() (*in module numpy.polynomial.hermite_e*), 1119
hermetrim() (*in module numpy.polynomial.hermite_e*), 1136
hermeval() (*in module numpy.polynomial.hermite_e*), 1122
hermeval2d() (*in module numpy.polynomial.hermite_e*), 1123
hermeval3d() (*in module numpy.polynomial.hermite_e*), 1124
hermevander() (*in module numpy.polynomial.hermite_e*), 1130
hermevander2d() (*in module numpy.polynomial.hermite_e*), 1131
hermevander3d() (*in module numpy.polynomial.hermite_e*), 1132
hermeweight() (*in module numpy.polynomial.hermite_e*), 1133
hermex (*in module numpy.polynomial.hermite_e*), 1118
hermezero (*in module numpy.polynomial.hermite_e*), 1118
hermfit() (*in module numpy.polynomial.hermite*), 1103
hermfromroots() (*in module numpy.polynomial.hermite*), 1097
hermgauss() (*in module numpy.polynomial.hermite*), 1101
hermgrid2d() (*in module numpy.polynomial.hermite*), 1093
hermgrid3d() (*in module numpy.polynomial.hermite*), 1094
hermint() (*in module numpy.polynomial.hermite*), 1096
Hermite (*class in numpy.polynomial.hermite*), 1076
HermiteE (*class in numpy.polynomial.hermite_e*), 1108
hermline() (*in module numpy.polynomial.hermite*), 1106
hermmul() (*in module numpy.polynomial.hermite*), 1088
hermmulx() (*in module numpy.polynomial.hermite*), 1087
hermone (*in module numpy.polynomial.hermite*), 1086
hermpow() (*in module numpy.polynomial.hermite*), 1089
hermroots() (*in module numpy.polynomial.hermite*), 1098
hermsub() (*in module numpy.polynomial.hermite*), 1087
hermtrim() (*in module numpy.polynomial.hermite*), 1105
hermval() (*in module numpy.polynomial.hermite*), 1090
hermval2d() (*in module numpy.polynomial.hermite*), 1091
hermval3d() (*in module numpy.polynomial.hermite*), 1092
hermvander() (*in module numpy.polynomial.hermite*),

1099
hermvander2d() (in module numpy.polynomial.hermite), 1100
hermvander3d() (in module numpy.polynomial.hermite), 1101
hermweight() (in module numpy.polynomial.hermite), 1102
hermx (in module numpy.polynomial.hermite), 1086
hermzero (in module numpy.polynomial.hermite), 1086
hfft() (in module numpy.fft), 685
histogram() (in module numpy), 1541
histogram2d() (in module numpy), 1543
histogram_bin_edges() (in module numpy), 1549
histogramdd() (in module numpy), 1546
hsplit (in module numpy.ma), 351
hsplit() (in module numpy), 546
hstack (in module numpy.ma), 349
hstack() (in module numpy), 540
hypergeometric() (in module numpy.random), 1383
hypergeometric() (numpy.random.Generator method), 1247
hypergeometric() (numpy.random.RandomState method), 1320
hypot (in module numpy), 882

## I

I() (numpy.matrix property), 118
i0() (in module numpy), 933
identity (in module numpy.ma), 395
identity (numpy.ufunc attribute), 450
identity() (in module numpy), 469
identity() (in module numpy.matlib), 989
identity() (numpy.polynomial.chebyshev.Chebyshev class method), 1048
identity() (numpy.polynomial.hermite_e.HermiteE class method), 1115
identity() (numpy.polynomial.hermite.Hermite class method), 1083
identity() (numpy.polynomial.laguerre.Laguerre class method), 1146
identity() (numpy.polynomial.legendre.Legendre class method), 1178
identity() (numpy.polynomial.polynomial.Polynomial class method), 1018
ids() (numpy.ma.MaskedArray method), 310
ifft() (in module numpy.fft), 667
ifft2() (in module numpy.fft), 671
ifftn() (in module numpy.fft), 674
ifftshift() (in module numpy.fft), 690
ihfft() (in module numpy.fft), 687
iinfo (class in numpy), 649
imag (numpy.generic attribute), 61
imag (numpy.ndarray attribute), 43
imag() (in module numpy), 961
imag() (numpy.ma.MaskedArray property), 267
import_array (C function), 1682
import_ufunc (C function), 1711
in1d() (in module numpy), 1478
index() (in module numpy.char), 595
index() (numpy.chararray method), 173
index() (numpy.char.chararray method), 612
indexing, 91, 99, 1725
indices() (in module numpy), 722
indices() (in module numpy.ma), 416
Inf (in module numpy), 431
inf (in module numpy), 435
Infinity (in module numpy), 431
info() (in module numpy), 714
infty (in module numpy), 435
inner() (in module numpy), 797
inner() (in module numpy.ma), 396
innerproduct() (in module numpy.ma), 397
insert() (in module numpy), 551
integ() (numpy.poly1d method), 1211
integ() (numpy.polynomial.chebyshev.Chebyshev method), 1048
integ() (numpy.polynomial.hermite_e.HermiteE method), 1115
integ() (numpy.polynomial.hermite.Hermite method), 1083
integ() (numpy.polynomial.laguerre.Laguerre method), 1146
integ() (numpy.polynomial.legendre.Legendre method), 1178
integ() (numpy.polynomial.polynomial.Polynomial method), 1018
integers() (numpy.random.Generator method), 1228
interface
    array, 419
interp() (in module numpy), 982
interpolate() (numpy.polynomial.chebyshev.Chebyshev class method), 1049
intersect1d() (in module numpy), 1480
inv() (in module numpy.linalg), 841
invert (in module numpy), 567
ipmt() (in module numpy), 699
irfft() (in module numpy.fft), 679
irfft2() (in module numpy.fft), 681
irfftn() (in module numpy.fft), 683
irr() (in module numpy), 701
is_busday() (in module numpy), 635
is_local_src_dir() (in module numpy.distutils.misc_util), 1596
is_mask() (in module numpy.ma), 336
is_masked() (in module numpy.ma), 335
is_sequence() (in module numpy.distutils.misc_util), 1596

is_string() (*in module numpy.distutils.misc_util*), 1596
isalnum() (*in module numpy.char*), 596
isalnum() (*numpy.chararray method*), 174
isalnum() (*numpy.char.chararray method*), 612
isalpha() (*in module numpy.char*), 596
isalpha() (*numpy.chararray method*), 174
isalpha() (*numpy.char.chararray method*), 613
isbuiltin (*numpy.dtype attribute*), 89
isclose() (*in module numpy*), 865
iscomplex() (*in module numpy*), 855
iscomplexobj() (*in module numpy*), 856
iscontiguous() (*numpy.ma.MaskedArray method*), 311
isdecimal() (*in module numpy.char*), 596
isdecimal() (*numpy.chararray method*), 174
isdecimal() (*numpy.char.chararray method*), 613
isdigit() (*in module numpy.char*), 597
isdigit() (*numpy.chararray method*), 174
isdigit() (*numpy.char.chararray method*), 613
isfinite (*in module numpy*), 849
isfortran() (*in module numpy*), 856
isin() (*in module numpy*), 1481
isinf (*in module numpy*), 850
islower() (*in module numpy.char*), 597
islower() (*numpy.chararray method*), 174
islower() (*numpy.char.chararray method*), 613
isnan (*in module numpy*), 851
isnat (*in module numpy*), 852
isnative (*numpy.dtype attribute*), 90
isneginf() (*in module numpy*), 853
isnumeric() (*in module numpy.char*), 598
isnumeric() (*numpy.chararray method*), 174
isnumeric() (*numpy.char.chararray method*), 613
isposinf() (*in module numpy*), 854
isreal() (*in module numpy*), 857
isrealobj() (*in module numpy*), 858
isscalar() (*in module numpy*), 858
issctype() (*in module numpy*), 652
isspace() (*in module numpy.char*), 598
isspace() (*numpy.chararray method*), 174
isspace() (*numpy.char.chararray method*), 613
issubclass_() (*in module numpy*), 654
issubdtype() (*in module numpy*), 653
issubsctype() (*in module numpy*), 653
istitle() (*in module numpy.char*), 598
istitle() (*numpy.chararray method*), 175
istitle() (*numpy.char.chararray method*), 613
isupper() (*in module numpy.char*), 599
isupper() (*numpy.chararray method*), 175
isupper() (*numpy.char.chararray method*), 614
item() (*numpy.chararray method*), 175
item() (*numpy.char.chararray method*), 614
item() (*numpy.generic method*), 67
item() (*numpy.ma.MaskedArray method*), 287
item() (*numpy.matrix method*), 136
item() (*numpy.ndarray method*), 17
item() (*numpy.recarray method*), 205
item() (*numpy.record method*), 228
itemset() (*numpy.generic method*), 67
itemset() (*numpy.matrix method*), 137
itemset() (*numpy.ndarray method*), 18
itemset() (*numpy.recarray method*), 206
itemset() (*numpy.record method*), 228
itemsize (*numpy.dtype attribute*), 85
itemsize (*numpy.generic attribute*), 61
itemsize (*numpy.ma.MaskedArray attribute*), 264
itemsize (*numpy.ndarray attribute*), 40
iterator
    C-API, 1689, 1705
iternext() (*numpy.nditer method*), 756
ix_() (*in module numpy*), 723

## J

join() (*in module numpy.char*), 580
join() (*numpy.chararray method*), 176
join() (*numpy.char.chararray method*), 615
jumped() (*numpy.random.MT19937 method*), 1440
jumped() (*numpy.random.PCG64 method*), 1444
jumped() (*numpy.random.Philox method*), 1448

## K

kaiser() (*in module numpy*), 1588
keyword arguments
    ufunc, 446
kind (*numpy.dtype attribute*), 84
knownfailureif() (*in module numpy.testing.dec*), 1567
kron() (*in module numpy*), 814

## L

lag2poly() (*in module numpy.polynomial.laguerre*), 1169
lagadd() (*in module numpy.polynomial.laguerre*), 1149
lagcompanion() (*in module numpy.polynomial.laguerre*), 1165
lagder() (*in module numpy.polynomial.laguerre*), 1158
lagdiv() (*in module numpy.polynomial.laguerre*), 1152
lagdomain (*in module numpy.polynomial.laguerre*), 1149
lagfit() (*in module numpy.polynomial.laguerre*), 1166
lagfromroots() (*in module numpy.polynomial.laguerre*), 1160
laggauss() (*in module numpy.polynomial.laguerre*), 1164
laggrid2d() (*in module numpy.polynomial.laguerre*), 1156

`laggrid3d()` (*in module numpy.polynomial.laguerre*), 1157
`lagint()` (*in module numpy.polynomial.laguerre*), 1159
`lagline()` (*in module numpy.polynomial.laguerre*), 1169
`lagmul()` (*in module numpy.polynomial.laguerre*), 1151
`lagmulx()` (*in module numpy.polynomial.laguerre*), 1151
`lagone` (*in module numpy.polynomial.laguerre*), 1149
`lagpow()` (*in module numpy.polynomial.laguerre*), 1153
`lagroots()` (*in module numpy.polynomial.laguerre*), 1161
`lagsub()` (*in module numpy.polynomial.laguerre*), 1150
`lagtrim()` (*in module numpy.polynomial.laguerre*), 1168
`Laguerre` (*class in numpy.polynomial.laguerre*), 1139
`lagval()` (*in module numpy.polynomial.laguerre*), 1153
`lagval2d()` (*in module numpy.polynomial.laguerre*), 1154
`lagval3d()` (*in module numpy.polynomial.laguerre*), 1155
`lagvander()` (*in module numpy.polynomial.laguerre*), 1162
`lagvander2d()` (*in module numpy.polynomial.laguerre*), 1163
`lagvander3d()` (*in module numpy.polynomial.laguerre*), 1164
`lagweight()` (*in module numpy.polynomial.laguerre*), 1165
`lagx` (*in module numpy.polynomial.laguerre*), 1149
`lagzero` (*in module numpy.polynomial.laguerre*), 1149
`laplace()` (*in module numpy.random*), 1385
`laplace()` (*numpy.random.Generator method*), 1249
`laplace()` (*numpy.random.RandomState method*), 1322
`lcm` (*in module numpy*), 941
`ldexp` (*in module numpy*), 938
`left_shift` (*in module numpy*), 569
`leg2poly()` (*in module numpy.polynomial.legendre*), 1201
`legadd()` (*in module numpy.polynomial.legendre*), 1181
`legcompanion()` (*in module numpy.polynomial.legendre*), 1197
`legder()` (*in module numpy.polynomial.legendre*), 1189
`legdiv()` (*in module numpy.polynomial.legendre*), 1184
`legdomain` (*in module numpy.polynomial.legendre*), 1181
`Legendre` (*class in numpy.polynomial.legendre*), 1171
`legfit()` (*in module numpy.polynomial.legendre*), 1197
`legfromroots()` (*in module numpy.polynomial.legendre*), 1192
`leggauss()` (*in module numpy.polynomial.legendre*), 1196
`leggrid2d()` (*in module numpy.polynomial.legendre*), 1188
`leggrid3d()` (*in module numpy.polynomial.legendre*), 1188
`legint()` (*in module numpy.polynomial.legendre*), 1190
`legline()` (*in module numpy.polynomial.legendre*), 1200
`legmul()` (*in module numpy.polynomial.legendre*), 1183
`legmulx()` (*in module numpy.polynomial.legendre*), 1182
`legone` (*in module numpy.polynomial.legendre*), 1181
`legpow()` (*in module numpy.polynomial.legendre*), 1184
`legroots()` (*in module numpy.polynomial.legendre*), 1193
`legsub()` (*in module numpy.polynomial.legendre*), 1182
`legtrim()` (*in module numpy.polynomial.legendre*), 1199
`legval()` (*in module numpy.polynomial.legendre*), 1185
`legval2d()` (*in module numpy.polynomial.legendre*), 1186
`legval3d()` (*in module numpy.polynomial.legendre*), 1187
`legvander()` (*in module numpy.polynomial.legendre*), 1194
`legvander2d()` (*in module numpy.polynomial.legendre*), 1194
`legvander3d()` (*in module numpy.polynomial.legendre*), 1195
`legweight()` (*in module numpy.polynomial.legendre*), 1196
`legx` (*in module numpy.polynomial.legendre*), 1181
`legzero` (*in module numpy.polynomial.legendre*), 1181
`less` (*in module numpy*), 870
`less()` (*in module numpy.char*), 591
`less_equal` (*in module numpy*), 870
`less_equal()` (*in module numpy.char*), 591
`lexsort()` (*in module numpy*), 1487
`LinAlgError`, 845
`linspace()` (*in module numpy*), 497
`linspace()` (*numpy.polynomial.chebyshev.Chebyshev method*), 1050
`linspace()` (*numpy.polynomial.hermite_e.HermiteE method*), 1116
`linspace()` (*numpy.polynomial.hermite.Hermite method*), 1084
`linspace()` (*numpy.polynomial.laguerre.Laguerre method*), 1147
`linspace()` (*numpy.polynomial.legendre.Legendre method*), 1179
`linspace()` (*numpy.polynomial.polynomial.Polynomial method*), 1019
`ljust()` (*in module numpy.char*), 580
`ljust()` (*numpy.chararray method*), 176
`ljust()` (*numpy.char.chararray method*), 615
`load()` (*in module numpy*), 761
`load_library()` (*in module numpy.ctypeslib*), 630

loadtxt() (*in module numpy*), 488
log (*in module numpy*), 926
log1p (*in module numpy*), 929
log2 (*in module numpy*), 928
log10 (*in module numpy*), 927
logaddexp (*in module numpy*), 930
logaddexp2 (*in module numpy*), 931
logical_and (*in module numpy*), 860
logical_not (*in module numpy*), 861
logical_or (*in module numpy*), 860
logical_xor (*in module numpy*), 862
logistic() (*in module numpy.random*), 1386
logistic() (*numpy.random.Generator method*), 1250
logistic() (*numpy.random.RandomState method*), 1323
lognormal() (*in module numpy.random*), 1388
lognormal() (*numpy.random.Generator method*), 1252
lognormal() (*numpy.random.RandomState method*), 1325
logseries() (*in module numpy.random*), 1391
logseries() (*numpy.random.Generator method*), 1254
logseries() (*numpy.random.RandomState method*), 1328
logspace() (*in module numpy*), 498
lookfor() (*in module numpy*), 713
lower() (*in module numpy.char*), 581
lower() (*numpy.chararray method*), 176
lower() (*numpy.char.chararray method*), 615
lstrip() (*in module numpy.char*), 581
lstrip() (*numpy.chararray method*), 176
lstrip() (*numpy.char.chararray method*), 615
lstsq() (*in module numpy.linalg*), 839

## M

MachAr (*class in numpy*), 650
make_config_py() (*numpy.distutils.misc_util.Configuration method*), 1610
make_mask() (*in module numpy.ma*), 356
make_mask_descr() (*in module numpy.ma*), 359
make_mask_none() (*in module numpy.ma*), 357
make_svn_version_py() (*numpy.distutils.misc_util.Configuration method*), 1610
mapdomain() (*in module numpy.polynomial.polyutils*), 1206
mapparms() (*in module numpy.polynomial.polyutils*), 1207
mapparms() (*numpy.polynomial.chebyshev.Chebyshev method*), 1050
mapparms() (*numpy.polynomial.hermite_e.HermiteE method*), 1116
mapparms() (*numpy.polynomial.hermite.Hermite method*), 1084
mapparms() (*numpy.polynomial.laguerre.Laguerre method*), 1147
mapparms() (*numpy.polynomial.legendre.Legendre method*), 1179
mapparms() (*numpy.polynomial.polynomial.Polynomial method*), 1019
mask (*numpy.ma.MaskedArray attribute*), 259
mask() (*numpy.ma.masked_array property*), 360
mask_cols() (*in module numpy.ma*), 365
mask_indices() (*in module numpy*), 728
mask_or() (*in module numpy.ma*), 358
mask_rowcols() (*in module numpy.ma*), 366
mask_rows() (*in module numpy.ma*), 367
masked (*in module numpy.ma*), 258
masked arrays, 237
masked_all() (*in module numpy.ma*), 322
masked_all_like() (*in module numpy.ma*), 323
masked_array (*in module numpy.ma*), 241
masked_equal() (*in module numpy.ma*), 244
masked_greater() (*in module numpy.ma*), 244
masked_greater_equal() (*in module numpy.ma*), 245
masked_inside() (*in module numpy.ma*), 245
masked_invalid() (*in module numpy.ma*), 246
masked_less() (*in module numpy.ma*), 246
masked_less_equal() (*in module numpy.ma*), 247
masked_not_equal() (*in module numpy.ma*), 247
masked_object() (*in module numpy.ma*), 247
masked_outside() (*in module numpy.ma*), 249
masked_print_options (*in module numpy.ma*), 258
masked_values() (*in module numpy.ma*), 249
masked_where() (*in module numpy.ma*), 251
MaskedArray (*class in numpy.ma*), 259
MaskType (*in module numpy.ma*), 317
mat() (*in module numpy*), 512
matmul (*in module numpy*), 800
matrix, 50, 116
matrix (*class in numpy*), 119
matrix_power() (*in module numpy.linalg*), 813
matrix_rank() (*in module numpy.linalg*), 833
max() (*in module numpy.ma*), 389
max() (*numpy.generic method*), 68
max() (*numpy.ma.MaskedArray method*), 298
max() (*numpy.matrix method*), 138
max() (*numpy.ndarray method*), 19
max() (*numpy.recarray method*), 206
max() (*numpy.record method*), 228
maximum (*in module numpy*), 974
maximum_fill_value() (*in module numpy.ma*), 373
maximum_sctype() (*in module numpy*), 658
may_share_memory() (*in module numpy*), 993
mean (*in module numpy.ma*), 383
mean() (*in module numpy*), 1525
mean() (*numpy.generic method*), 68

mean() (*numpy.ma.MaskedArray method*), 298
mean() (*numpy.matrix method*), 139
mean() (*numpy.ndarray method*), 19
mean() (*numpy.recarray method*), 206
mean() (*numpy.record method*), 228
median() (*in module numpy*), 1522
median() (*in module numpy.ma*), 383
memmap (*class in numpy*), 160
memory maps, 159
memory model
    ndarray, 1723
meshgrid() (*in module numpy*), 502
methods
    accumulate, ufunc, 1729
    reduce, ufunc, 1729
    reduceat, ufunc, 1730
    ufunc, 452
mgrid (*in module numpy*), 505
min() (*in module numpy.ma*), 390
min() (*numpy.generic method*), 68
min() (*numpy.ma.MaskedArray method*), 299
min() (*numpy.matrix method*), 139
min() (*numpy.ndarray method*), 19
min() (*numpy.recarray method*), 207
min() (*numpy.record method*), 228
min_scalar_type() (*in module numpy*), 643
mingw32() (*in module numpy.distutils.misc_util*), 1596
minimum (*in module numpy*), 976
minrelpath() (*in module numpy.distutils.misc_util*), 1596
mintypecode() (*in module numpy*), 657
mirr() (*in module numpy*), 702
mod (*in module numpy*), 955
mod() (*in module numpy.char*), 577
modf (*in module numpy*), 956
moveaxis() (*in module numpy*), 518
mr_ (*in module numpy.ma*), 352
msort() (*in module numpy*), 1491
MT19937 (*class in numpy.random*), 1439
multi_dot() (*in module numpy.linalg*), 795
multinomial() (*in module numpy.random*), 1392
multinomial() (*numpy.random.Generator method*), 1256
multinomial() (*numpy.random.RandomState method*), 1329
multiply (*in module numpy*), 946
multiply() (*in module numpy.char*), 576
multivariate_hypergeometric() (*numpy.random.Generator method*), 1257
multivariate_normal() (*in module numpy.random*), 1394
multivariate_normal() (*numpy.random.Generator method*), 1259
multivariate_normal() (*numpy.random.RandomState method*), 1331

# N

name (*numpy.dtype attribute*), 85
names (*numpy.dtype attribute*), 87
NAN (*in module numpy*), 431
NaN (*in module numpy*), 433
nan (*in module numpy*), 436
nan_to_num() (*in module numpy*), 980
nanargmax() (*in module numpy*), 1496
nanargmin() (*in module numpy*), 1498
nancumprod() (*in module numpy*), 912
nancumsum() (*in module numpy*), 913
nanmax() (*in module numpy*), 1510
nanmean() (*in module numpy*), 1532
nanmedian() (*in module numpy*), 1531
nanmin() (*in module numpy*), 1508
nanpercentile() (*in module numpy*), 1516
nanprod() (*in module numpy*), 907
nanquantile() (*in module numpy*), 1520
nanstd() (*in module numpy*), 1533
nansum() (*in module numpy*), 908
nanvar() (*in module numpy*), 1535
nargs (*numpy.ufunc attribute*), 449
nbytes (*numpy.ma.MaskedArray attribute*), 265
nbytes (*numpy.ndarray attribute*), 41
ndarray, 99
    C-API, 1647, 1689
    memory model, 1723
    special methods getitem, 92
    special methods setitem, 92
    view, 94
ndarray (*class in numpy*), 4
NDArrayOperatorsMixin (*class in numpy.lib.mixins*), 995
ndenumerate (*class in numpy*), 234
ndim (*numpy.generic attribute*), 61
ndim (*numpy.ma.MaskedArray attribute*), 265
ndim (*numpy.ndarray attribute*), 40
ndincr() (*numpy.ndindex method*), 757
ndindex (*class in numpy*), 757
nditer (*class in numpy*), 750
ndpointer() (*in module numpy.ctypeslib*), 631
negative (*in module numpy*), 945
negative_binomial() (*in module numpy.random*), 1396
negative_binomial() (*numpy.random.Generator method*), 1261
negative_binomial() (*numpy.random.RandomState method*), 1333
nested_iters() (*in module numpy*), 757
newaxis, 92
newaxis (*in module numpy*), 436

newbyteorder() (*numpy.dtype method*), 77
newbyteorder() (*numpy.generic method*), 68
newbyteorder() (*numpy.matrix method*), 140
newbyteorder() (*numpy.ndarray method*), 19
newbyteorder() (*numpy.recarray method*), 207
newbyteorder() (*numpy.record method*), 228
nextafter (*in module numpy*), 939
nin (*numpy.ufunc attribute*), 448
NINF (*in module numpy*), 431
njoin() (*in module numpy.distutils.misc_util*), 1596
NO_IMPORT_ARRAY (*C macro*), 1682
NO_IMPORT_UFUNC (*C variable*), 1711
nomask (*in module numpy.ma*), 258
noncentral_chisquare() (*in module numpy.random*), 1397
noncentral_chisquare() (*numpy.random.Generator method*), 1262
noncentral_chisquare() (*numpy.random.RandomState method*), 1334
noncentral_f() (*in module numpy.random*), 1398
noncentral_f() (*numpy.random.Generator method*), 1263
noncentral_f() (*numpy.random.RandomState method*), 1337
non-contiguous, 35
nonzero (*in module numpy.ma*), 332
nonzero() (*in module numpy*), 719
nonzero() (*numpy.chararray method*), 176
nonzero() (*numpy.char.chararray method*), 615
nonzero() (*numpy.generic method*), 69
nonzero() (*numpy.ma.MaskedArray method*), 288
nonzero() (*numpy.matrix method*), 140
nonzero() (*numpy.ndarray method*), 20
nonzero() (*numpy.recarray method*), 207
nonzero() (*numpy.record method*), 229
norm() (*in module numpy.linalg*), 828
normal() (*in module numpy.random*), 1401
normal() (*numpy.random.Generator method*), 1266
normal() (*numpy.random.RandomState method*), 1338
not_equal (*in module numpy*), 872
not_equal() (*in module numpy.char*), 590
notmasked_contiguous() (*in module numpy.ma*), 362
notmasked_edges() (*in module numpy.ma*), 363
nout (*numpy.ufunc attribute*), 448
nper() (*in module numpy*), 703
npv() (*in module numpy*), 696
NPY_1_PI (*C variable*), 1716
NPY_2_PI (*C variable*), 1716
NPY_ALLOW_C_API (*C macro*), 1685
NPY_ALLOW_C_API_DEF (*C macro*), 1685
NPY_ANYORDER (*C variable*), 1688
NPY_ARRAY_ALIGNED (*C variable*), 1652, 1663
NPY_ARRAY_BEHAVED (*C variable*), 1653, 1664
NPY_ARRAY_BEHAVED_NS (*C variable*), 1654, 1664
NPY_ARRAY_C_CONTIGUOUS (*C variable*), 1652, 1663
NPY_ARRAY_CARRAY (*C variable*), 1653, 1664
NPY_ARRAY_CARRAY_RO (*C variable*), 1653, 1664
NPY_ARRAY_DEFAULT (*C variable*), 1653, 1664
NPY_ARRAY_ELEMENTSTRIDES (*C variable*), 1654
NPY_ARRAY_ENSUREARRAY (*C variable*), 1652, 1664
NPY_ARRAY_ENSURECOPY (*C variable*), 1652, 1664
NPY_ARRAY_F_CONTIGUOUS (*C variable*), 1652, 1663
NPY_ARRAY_FARRAY (*C variable*), 1653, 1664
NPY_ARRAY_FARRAY_RO (*C variable*), 1653, 1664
NPY_ARRAY_FORCECAST (*C variable*), 1652, 1664
NPY_ARRAY_IN_ARRAY (*C variable*), 1653
NPY_ARRAY_IN_FARRAY (*C variable*), 1653
NPY_ARRAY_INOUT_ARRAY (*C variable*), 1653
NPY_ARRAY_INOUT_FARRAY (*C variable*), 1653
NPY_ARRAY_NOTSWAPPED (*C variable*), 1654, 1664
NPY_ARRAY_OUT_ARRAY (*C variable*), 1653
NPY_ARRAY_OUT_FARRAY (*C variable*), 1653
NPY_ARRAY_OWNDATA (*C variable*), 1663
NPY_ARRAY_UPDATE_ALL (*C variable*), 1664
NPY_ARRAY_UPDATEIFCOPY (*C variable*), 1652, 1663
NPY_ARRAY_WRITEABLE (*C variable*), 1652, 1663
NPY_ARRAY_WRITEBACKIFCOPY (*C variable*), 1652, 1663
NPY_AUXDATA_CLONE (*C function*), 1675
NPY_AUXDATA_FREE (*C function*), 1674
NPY_BEGIN_ALLOW_THREADS (*C macro*), 1685
NPY_BEGIN_THREADS (*C macro*), 1685
NPY_BEGIN_THREADS_DEF (*C macro*), 1685
NPY_BEGIN_THREADS_DESCR (*C function*), 1685
NPY_BEGIN_THREADS_THRESHOLDED (*C function*), 1685
NPY_BIG_ENDIAN (*C variable*), 1641
npy_bool (*C type*), 1646
NPY_BOOL (*C variable*), 1642
NPY_BOOL_SCALAR (*C variable*), 1688
NPY_BUFSIZE (*C variable*), 1686
NPY_BYTE (*C variable*), 1642
NPY_BYTE_ORDER (*C variable*), 1641
NPY_CASTING (*C type*), 1688
NPY_CDOUBLE (*C variable*), 1643
NPY_CFLOAT (*C variable*), 1643
npy_clear_floatstatus (*C function*), 1717
npy_clear_floatstatus_barrier (*C function*), 1717
NPY_CLIP (*C variable*), 1668, 1688
NPY_CLIPMODE (*C type*), 1688
NPY_CLONGDOUBLE (*C variable*), 1643
NPY_COMPLEX64 (*C variable*), 1643
NPY_COMPLEX128 (*C variable*), 1643
NPY_COMPLEX_SCALAR (*C variable*), 1688
npy_copysign (*C function*), 1715
NPY_CORDER (*C variable*), 1688

NPY_CPU_AMD64 (*C variable*), 1641
NPY_CPU_IA64 (*C variable*), 1641
NPY_CPU_PARISC (*C variable*), 1641
NPY_CPU_PPC (*C variable*), 1641
NPY_CPU_PPC64 (*C variable*), 1641
NPY_CPU_S390 (*C variable*), 1641
NPY_CPU_SPARC (*C variable*), 1641
NPY_CPU_SPARC64 (*C variable*), 1641
NPY_CPU_X86 (*C variable*), 1641
NPY_DATETIME (*C variable*), 1643
NPY_DEFAULT_TYPE (*C variable*), 1644
NPY_DISABLE_C_API (*C macro*), 1685
NPY_DOUBLE (*C variable*), 1643
npy_double_to_half (*C function*), 1719
npy_doublebits_to_halfbits (*C function*), 1720
NPY_E (*C variable*), 1716
NPY_END_ALLOW_THREADS (*C macro*), 1685
NPY_END_THREADS (*C macro*), 1685
NPY_END_THREADS_DESCR (*C function*), 1685
NPY_EQUIV_CASTING (*C variable*), 1688
NPY_EULER (*C variable*), 1716
NPY_FAIL (*C variable*), 1686
NPY_FALSE (*C variable*), 1686
NPY_FLOAT (*C variable*), 1643
NPY_FLOAT16 (*C variable*), 1643
NPY_FLOAT32 (*C variable*), 1643
NPY_FLOAT64 (*C variable*), 1643
NPY_FLOAT_SCALAR (*C variable*), 1688
npy_float_to_half (*C function*), 1719
npy_floatbits_to_halfbits (*C function*), 1720
NPY_FORTRANORDER (*C variable*), 1688
NPY_FROM_FIELDS (*C variable*), 1627
npy_get_floatstatus (*C function*), 1717
npy_get_floatstatus_barrier (*C function*), 1717
npy_half (*C type*), 1647
NPY_HALF (*C variable*), 1643
npy_half_copysign (*C function*), 1720
npy_half_eq (*C function*), 1719
npy_half_eq_nonan (*C function*), 1719
npy_half_ge (*C function*), 1719
npy_half_gt (*C function*), 1719
npy_half_isfinite (*C function*), 1720
npy_half_isinf (*C function*), 1720
npy_half_isnan (*C function*), 1720
npy_half_iszero (*C function*), 1720
npy_half_le (*C function*), 1719
npy_half_le_nonan (*C function*), 1720
npy_half_lt (*C function*), 1719
npy_half_lt_nonan (*C function*), 1720
NPY_HALF_NAN (*C variable*), 1719
npy_half_ne (*C function*), 1719
NPY_HALF_NEGONE (*C variable*), 1719
npy_half_nextafter (*C function*), 1720
NPY_HALF_NINF (*C variable*), 1719
NPY_HALF_NZERO (*C variable*), 1719
NPY_HALF_ONE (*C variable*), 1719
NPY_HALF_PINF (*C variable*), 1719
NPY_HALF_PZERO (*C variable*), 1719
npy_half_signbit (*C function*), 1720
npy_half_spacing (*C function*), 1720
npy_half_to_double (*C function*), 1719
npy_half_to_float (*C function*), 1719
NPY_HALF_ZERO (*C variable*), 1719
npy_halfbits_to_doublebits (*C function*), 1720
npy_halfbits_to_floatbits (*C function*), 1720
NPY_HEAPSORT (*C variable*), 1687
NPY_INFINITY (*C variable*), 1715
npy_int (*C type*), 1646
NPY_INT (*C variable*), 1642
NPY_INT8 (*C variable*), 1642
npy_int16 (*C type*), 1646
NPY_INT16 (*C variable*), 1642
npy_int32 (*C type*), 1646
NPY_INT32 (*C variable*), 1642
npy_int64 (*C type*), 1646
NPY_INT64 (*C variable*), 1643
NPY_INTERRUPT_H (*C variable*), 1642
NPY_INTNEG_SCALAR (*C variable*), 1688
npy_intp (*C type*), 1646
NPY_INTP (*C variable*), 1644
NPY_INTPOS_SCALAR (*C variable*), 1688
npy_isfinite (*C function*), 1715
npy_isinf (*C function*), 1715
npy_isnan (*C function*), 1715
NPY_ITEM_HASOBJECT (*C variable*), 1626
NPY_ITEM_IS_POINTER (*C variable*), 1627
NPY_ITEM_REFCOUNT (*C variable*), 1626
NPY_ITER_ALIGNED (*C variable*), 1696
NPY_ITER_ALLOCATE (*C variable*), 1696
NPY_ITER_ARRAYMASK (*C variable*), 1696
NPY_ITER_BUFFERED (*C variable*), 1695
NPY_ITER_C_INDEX (*C variable*), 1693
NPY_ITER_COMMON_DTYPE (*C variable*), 1694
NPY_ITER_CONTIG (*C variable*), 1696
NPY_ITER_COPY (*C variable*), 1696
NPY_ITER_COPY_IF_OVERLAP (*C variable*), 1695
NPY_ITER_DELAY_BUFALLOC (*C variable*), 1695
NPY_ITER_DONT_NEGATE_STRIDES (*C variable*), 1694
NPY_ITER_EXTERNAL_LOOP (*C variable*), 1694
NPY_ITER_F_INDEX (*C variable*), 1693
NPY_ITER_GROWINNER (*C variable*), 1695
NPY_ITER_MULTI_INDEX (*C variable*), 1694
NPY_ITER_NBO (*C variable*), 1696
NPY_ITER_NO_BROADCAST (*C variable*), 1696
NPY_ITER_NO_SUBTYPE (*C variable*), 1696

NPY_ITER_OVERLAP_ASSUME_ELEMENTWISE (*C variable*), 1697
NPY_ITER_RANGED (*C variable*), 1694
NPY_ITER_READONLY (*C variable*), 1695
NPY_ITER_READWRITE (*C variable*), 1695
NPY_ITER_REDUCE_OK (*C variable*), 1694
NPY_ITER_REFS_OK (*C variable*), 1694
NPY_ITER_UPDATEIFCOPY (*C variable*), 1696
NPY_ITER_WRITEMASKED (*C variable*), 1697
NPY_ITER_WRITEONLY (*C variable*), 1695
NPY_ITER_ZEROSIZE_OK (*C variable*), 1694
NPY_KEEPORDER (*C variable*), 1688
NPY_LIKELY (*C variable*), 1642
NPY_LIST_PICKLE (*C variable*), 1627
NPY_LITTLE_ENDIAN (*C variable*), 1641
NPY_LOG2E (*C variable*), 1716
NPY_LOG10E (*C variable*), 1716
NPY_LOGE2 (*C variable*), 1716
NPY_LOGE10 (*C variable*), 1716
NPY_LONG (*C variable*), 1643
NPY_LONGDOUBLE (*C variable*), 1643
NPY_LONGLONG (*C variable*), 1643
NPY_LOOP_BEGIN_THREADS (*C macro*), 1707
NPY_LOOP_END_THREADS (*C macro*), 1707
NPY_MASK (*C variable*), 1644
NPY_MAX_BUFSIZE (*C variable*), 1686
NPY_MAXARGS (*C variable*), 1686
NPY_MAXDIMS (*C variable*), 1686
NPY_MERGESORT (*C variable*), 1687
NPY_MIN_BUFSIZE (*C variable*), 1686
NPY_NAN (*C variable*), 1715
NPY_NEEDS_INIT (*C variable*), 1627
NPY_NEEDS_PYAPI (*C variable*), 1627
NPY_NEIGHBORHOOD_ITER_CIRCULAR_PADDING (*C macro*), 1677
NPY_NEIGHBORHOOD_ITER_CONSTANT_PADDING (*C macro*), 1677
NPY_NEIGHBORHOOD_ITER_MIRROR_PADDING (*C macro*), 1677
NPY_NEIGHBORHOOD_ITER_ONE_PADDING (*C macro*), 1677
NPY_NEIGHBORHOOD_ITER_ZERO_PADDING (*C macro*), 1677
npy_nextafter (*C function*), 1716
NPY_NO_CASTING (*C variable*), 1688
NPY_NOSCALAR (*C variable*), 1688
NPY_NOTYPE (*C variable*), 1644
NPY_NSCALARKINDS (*C variable*), 1688
NPY_NSORTS (*C variable*), 1687
NPY_NTYPES (*C variable*), 1644
NPY_NUM_FLOATTYPE (*C variable*), 1686
NPY_NZERO (*C variable*), 1715
NPY_OBJECT (*C variable*), 1644
NPY_OBJECT_DTYPE_FLAGS (*C variable*), 1627
NPY_OBJECT_SCALAR (*C variable*), 1688
NPY_ORDER (*C type*), 1688
NPY_OUT_ARRAY (*C variable*), 1653
NPY_PI (*C variable*), 1716
NPY_PI_2 (*C variable*), 1716
NPY_PI_4 (*C variable*), 1716
NPY_PRIORITY (*C variable*), 1686
NPY_PZERO (*C variable*), 1715
NPY_QUICKSORT (*C variable*), 1687
NPY_RAISE (*C variable*), 1668, 1688
NPY_SAFE_CASTING (*C variable*), 1689
NPY_SAME_KIND_CASTING (*C variable*), 1689
NPY_SCALAR_PRIORITY (*C variable*), 1686
NPY_SCALARKIND (*C type*), 1688
npy_set_floatstatus_divbyzero (*C function*), 1716
npy_set_floatstatus_invalid (*C function*), 1717
npy_set_floatstatus_overflow (*C function*), 1716
npy_set_floatstatus_underflow (*C function*), 1717
npy_short (*C type*), 1646
NPY_SHORT (*C variable*), 1642
NPY_SIGINT_OFF (*C variable*), 1642
NPY_SIGINT_ON (*C variable*), 1642
NPY_SIGJMP_BUF (*C variable*), 1642
NPY_SIGLONGJMP (*C variable*), 1642
npy_signbit (*C function*), 1715
NPY_SIGSETJMP (*C variable*), 1642
NPY_SIZEOF_DOUBLE (*C variable*), 1641
NPY_SIZEOF_FLOAT (*C variable*), 1641
NPY_SIZEOF_INT (*C variable*), 1640
NPY_SIZEOF_LONG (*C variable*), 1640
NPY_SIZEOF_LONG_DOUBLE (*C variable*), 1641
NPY_SIZEOF_LONGLONG (*C variable*), 1640
NPY_SIZEOF_PY_INTPTR_T (*C variable*), 1641
NPY_SIZEOF_PY_LONG_LONG (*C variable*), 1641
NPY_SIZEOF_SHORT (*C variable*), 1640
NPY_SORTKIND (*C type*), 1687
npy_spacing (*C function*), 1716
NPY_STABLESORT (*C variable*), 1687
NPY_STRING (*C variable*), 1644
NPY_SUBTYPE_PRIORITY (*C variable*), 1686
NPY_SUCCEED (*C variable*), 1686
NPY_TIMEDELTA (*C variable*), 1644
NPY_TRUE (*C variable*), 1686
NPY_TYPES (*C variable*), 1642
NPY_UBYTE (*C variable*), 1643
npy_uint (*C type*), 1646
NPY_UINT (*C variable*), 1643
NPY_UINT8 (*C variable*), 1643
npy_uint16 (*C type*), 1646
NPY_UINT16 (*C variable*), 1643

npy_uint32 (*C type*), 1646
NPY_UINT32 (*C variable*), 1643
npy_uint64 (*C type*), 1646
NPY_UINT64 (*C variable*), 1643
npy_uintp (*C type*), 1646
NPY_UINTP (*C variable*), 1644
NPY_ULONG (*C variable*), 1643
NPY_ULONGLONG (*C variable*), 1643
NPY_UNICODE (*C variable*), 1644
NPY_UNLIKELY (*C variable*), 1642
NPY_UNSAFE_CASTING (*C variable*), 1689
NPY_UNUSED (*C variable*), 1642
NPY_USE_GETITEM (*C variable*), 1627
NPY_USE_SETITEM (*C variable*), 1627
NPY_USERDEF (*C variable*), 1644
npy_ushort (*C type*), 1646
NPY_USHORT (*C variable*), 1643
NPY_VERSION (*C variable*), 1686
NPY_VOID (*C variable*), 1644
NPY_WRAP (*C variable*), 1668, 1688
NpyAuxData (*C type*), 1673
NpyAuxData_CloneFunc (*C type*), 1674
NpyAuxData_FreeFunc (*C type*), 1674
NpyIter (*C type*), 1692
NpyIter_AdvancedNew (*C function*), 1697
NpyIter_Copy (*C function*), 1698
NpyIter_CreateCompatibleStrides (*C function*), 1702
NpyIter_Deallocate (*C function*), 1699
NpyIter_EnableExternalLoop (*C function*), 1699
NpyIter_GetAxisStrideArray (*C function*), 1701
NpyIter_GetBufferSize (*C function*), 1701
NpyIter_GetDataPtrArray (*C function*), 1704
NpyIter_GetDescrArray (*C function*), 1701
NpyIter_GetGetMultiIndex (*C function*), 1704
NpyIter_GetIndexPtr (*C function*), 1704
NpyIter_GetInitialDataPtrArray (*C function*), 1704
NpyIter_GetInnerFixedStrideArray (*C function*), 1705
NpyIter_GetInnerLoopSizePtr (*C function*), 1705
NpyIter_GetInnerStrideArray (*C function*), 1704
NpyIter_GetIterIndex (*C function*), 1700
NpyIter_GetIterIndexRange (*C function*), 1700
NpyIter_GetIterNext (*C function*), 1703
NpyIter_GetIterSize (*C function*), 1700
NpyIter_GetIterView (*C function*), 1702
NpyIter_GetMultiIndexFunc (*C type*), 1692
NpyIter_GetNDim (*C function*), 1701
NpyIter_GetNOp (*C function*), 1701
NpyIter_GetOperandArray (*C function*), 1701
NpyIter_GetReadFlags (*C function*), 1702
NpyIter_GetShape (*C function*), 1701
NpyIter_GetWriteFlags (*C function*), 1702
NpyIter_GotoIndex (*C function*), 1700
NpyIter_GotoIterIndex (*C function*), 1700
NpyIter_GotoMultiIndex (*C function*), 1700
NpyIter_HasDelayedBufAlloc (*C function*), 1701
NpyIter_HasExternalLoop (*C function*), 1701
NpyIter_HasIndex (*C function*), 1701
NpyIter_HasMultiIndex (*C function*), 1701
NpyIter_IsBuffered (*C function*), 1701
NpyIter_IsFirstVisit (*C function*), 1702
NpyIter_IsGrowInner (*C function*), 1701
NpyIter_IterNextFunc (*C type*), 1692
NpyIter_MultiNew (*C function*), 1693
NpyIter_New (*C function*), 1693
NpyIter_RemoveMultiIndex (*C function*), 1698
NpyIter_RequiresBuffering (*C function*), 1701
NpyIter_Reset (*C function*), 1699
NpyIter_ResetBasePointers (*C function*), 1699
NpyIter_ResetToIterIndexRange (*C function*), 1699
NpyIter_Type (*C type*), 1692
ntypes (*numpy.ufunc attribute*), 449
num (*numpy.dtype attribute*), 85
numpy (*module*), 1
numpy.char (*module*), 575
numpy.ctypeslib (*module*), 629
numpy.distutils (*module*), 1595
numpy.distutils.ccompiler (*module*), 1598
numpy.distutils.exec_command (*module*), 1600
numpy.distutils.misc_util (*module*), 1595
numpy.doc.constants (*module*), 431
numpy.doc.internals (*module*), 1731
numpy.dual (*module*), 659
numpy.fft (*module*), 665
numpy.lib.format (*module*), 791
numpy.lib.scimath (*module*), 660
numpy.linalg (*module*), 793
numpy.ma (*module*), 237
numpy.matlib (*module*), 985
numpy.polynomial.chebyshev (*module*), 1041
numpy.polynomial.hermite (*module*), 1076
numpy.polynomial.hermite_e (*module*), 1108
numpy.polynomial.laguerre (*module*), 1139
numpy.polynomial.legendre (*module*), 1171
numpy.polynomial.polynomial (*module*), 1011
numpy.polynomial.polyutils (*module*), 1202
numpy.random (*module*), 1224
numpy.testing (*module*), 1554
NumpyVersion (*class in numpy.lib*), 996
NZERO (*in module numpy*), 432

## O

obj2sctype() (*in module numpy*), 645

offset, 35
ogrid (*in module numpy*), 506
ones (*in module numpy.ma*), 324
ones() (*in module numpy*), 469
ones() (*in module numpy.matlib*), 987
ones_like() (*in module numpy*), 470
open() (*numpy.DataSource method*), 790
operation, 50, 305
operator, 50, 305
outer() (*in module numpy*), 799
outer() (*in module numpy.ma*), 398
outer() (*numpy.ufunc method*), 457
outerproduct() (*in module numpy.ma*), 400

P

packbits() (*in module numpy*), 571
pad() (*in module numpy*), 1001
pareto() (*in module numpy.random*), 1403
pareto() (*numpy.random.Generator method*), 1268
pareto() (*numpy.random.RandomState method*), 1340
partition() (*in module numpy*), 1491
partition() (*in module numpy.char*), 582
partition() (*numpy.matrix method*), 141
partition() (*numpy.ndarray method*), 20
partition() (*numpy.recarray method*), 208
paths() (*numpy.distutils.misc_util.Configuration method*), 1609
PCG64 (*class in numpy.random*), 1442
percentile() (*in module numpy*), 1513
permutation() (*in module numpy.random*), 1405
permutation() (*numpy.random.Generator method*), 1233
permutation() (*numpy.random.RandomState method*), 1305
Philox (*class in numpy.random*), 1445
pi (*in module numpy*), 437
piecewise() (*in module numpy*), 711
PINF (*in module numpy*), 433
pinv() (*in module numpy.linalg*), 842
place() (*in module numpy*), 744
pmt() (*in module numpy*), 697
poisson() (*in module numpy.random*), 1406
poisson() (*numpy.random.Generator method*), 1270
poisson() (*numpy.random.RandomState method*), 1342
poly() (*in module numpy*), 1212
poly1d (*class in numpy*), 1208
poly2cheb() (*in module numpy.polynomial.chebyshev*), 1074
poly2herm() (*in module numpy.polynomial.hermite*), 1107
poly2herme() (*in module numpy.polynomial.hermite_e*), 1138
poly2lag() (*in module numpy.polynomial.laguerre*), 1170
poly2leg() (*in module numpy.polynomial.legendre*), 1201
polyadd() (*in module numpy*), 1221
polyadd() (*in module numpy.polynomial.polynomial*), 1021
PolyBase (*class in numpy.polynomial.polyutils*), 1203
polycompanion() (*in module numpy.polynomial.polynomial*), 1037
polyder() (*in module numpy*), 1218
polyder() (*in module numpy.polynomial.polynomial*), 1029
polydiv() (*in module numpy*), 1221
polydiv() (*in module numpy.polynomial.polynomial*), 1023
polydomain (*in module numpy.polynomial.polynomial*), 1021
PolyDomainError, 1202
PolyError, 1202
polyfit() (*in module numpy*), 1215
polyfit() (*in module numpy.ma*), 403
polyfit() (*in module numpy.polynomial.polynomial*), 1037
polyfromroots() (*in module numpy.polynomial.polynomial*), 1031
polygrid2d() (*in module numpy.polynomial.polynomial*), 1027
polygrid3d() (*in module numpy.polynomial.polynomial*), 1028
polyint() (*in module numpy*), 1219
polyint() (*in module numpy.polynomial.polynomial*), 1030
polyline() (*in module numpy.polynomial.polynomial*), 1040
polymul() (*in module numpy*), 1222
polymul() (*in module numpy.polynomial.polynomial*), 1023
polymulx() (*in module numpy.polynomial.polynomial*), 1022
Polynomial (*class in numpy.polynomial.polynomial*), 1011
polyone (*in module numpy.polynomial.polynomial*), 1021
polypow() (*in module numpy.polynomial.polynomial*), 1024
polyroots() (*in module numpy.polynomial.polynomial*), 1032
polysub() (*in module numpy*), 1223
polysub() (*in module numpy.polynomial.polynomial*), 1022
polytrim() (*in module numpy.polynomial.polynomial*), 1039
polyval() (*in module numpy*), 1211
polyval() (*in module numpy.polynomial.polynomial*), 1024

`polyval2d()` (*in module numpy.polynomial.polynomial*), 1026
`polyval3d()` (*in module numpy.polynomial.polynomial*), 1027
`polyvalfromroots()` (*in module numpy.polynomial.polynomial*), 1033
`polyvander()` (*in module numpy.polynomial.polynomial*), 1034
`polyvander2d()` (*in module numpy.polynomial.polynomial*), 1035
`polyvander3d()` (*in module numpy.polynomial.polynomial*), 1036
`polyx` (*in module numpy.polynomial.polynomial*), 1021
`polyzero` (*in module numpy.polynomial.polynomial*), 1021
`positive` (*in module numpy*), 945
`power` (*in module numpy*), 948
`power()` (*in module numpy.ma*), 384
`power()` (*in module numpy.random*), 1408
`power()` (*numpy.random.Generator method*), 1271
`power()` (*numpy.random.RandomState method*), 1344
`ppmt()` (*in module numpy*), 699
`pprint()` (*numpy.record method*), 229
`printoptions()` (*in module numpy*), 786
`prod` (*in module numpy.ma*), 385
`prod()` (*in module numpy*), 902
`prod()` (*numpy.generic method*), 69
`prod()` (*numpy.ma.MaskedArray method*), 299
`prod()` (*numpy.matrix method*), 142
`prod()` (*numpy.ndarray method*), 21
`prod()` (*numpy.recarray method*), 209
`prod()` (*numpy.record method*), 229
`product()` (*numpy.ma.MaskedArray method*), 300
`promote_types()` (*in module numpy*), 642
`protocol`
  `array`, 419
`ptp()` (*in module numpy*), 1511
`ptp()` (*in module numpy.ma*), 391
`ptp()` (*numpy.generic method*), 69
`ptp()` (*numpy.ma.MaskedArray method*), 300
`ptp()` (*numpy.matrix method*), 142
`ptp()` (*numpy.ndarray method*), 21
`ptp()` (*numpy.recarray method*), 209
`ptp()` (*numpy.record method*), 229
`put()` (*in module numpy*), 745
`put()` (*numpy.chararray method*), 177
`put()` (*numpy.char.chararray method*), 616
`put()` (*numpy.generic method*), 69
`put()` (*numpy.ma.MaskedArray method*), 289
`put()` (*numpy.matrix method*), 143
`put()` (*numpy.ndarray method*), 21
`put()` (*numpy.recarray method*), 209
`put()` (*numpy.record method*), 229
`put_along_axis()` (*in module numpy*), 746
`putmask()` (*in module numpy*), 747
`pv()` (*in module numpy*), 694
`PY_ARRAY_UNIQUE_SYMBOL` (*C macro*), 1682
`PY_UFUNC_UNIQUE_SYMBOL` (*C variable*), 1711
`PyArray_All` (*C function*), 1671
`PyArray_Any` (*C function*), 1671
`PyArray_Arange` (*C function*), 1651
`PyArray_ArangeObj` (*C function*), 1651
`PyArray_ArgMax` (*C function*), 1670
`PyArray_ArgMin` (*C function*), 1670
`PyArray_ArgPartition` (*C function*), 1669
`PyArray_ArgSort` (*C function*), 1668
`PyArray_ArrayDescr.base` (*C member*), 1627
`PyArray_ArrayDescr.shape` (*C member*), 1628
`PyArray_ArrayType` (*C function*), 1660
`PyArray_ArrFuncs` (*C type*), 1628
`PyArray_ArrFuncs.argmax` (*C member*), 1629
`PyArray_ArrFuncs.argmin` (*C member*), 1631
`PyArray_ArrFuncs.argsort` (*C member*), 1630
`PyArray_ArrFuncs.cancastscalarkindto` (*C member*), 1630
`PyArray_ArrFuncs.cancastto` (*C member*), 1630
`PyArray_ArrFuncs.cast` (*C member*), 1629
`PyArray_ArrFuncs.castdict` (*C member*), 1630
`PyArray_ArrFuncs.compare` (*C member*), 1629
`PyArray_ArrFuncs.copyswap` (*C member*), 1629
`PyArray_ArrFuncs.copyswapn` (*C member*), 1629
`PyArray_ArrFuncs.dotfunc` (*C member*), 1629
`PyArray_ArrFuncs.fastclip` (*C member*), 1630
`PyArray_ArrFuncs.fastputmask` (*C member*), 1631
`PyArray_ArrFuncs.fasttake` (*C member*), 1631
`PyArray_ArrFuncs.fill` (*C member*), 1630
`PyArray_ArrFuncs.fillwithscalar` (*C member*), 1630
`PyArray_ArrFuncs.fromstr` (*C member*), 1630
`PyArray_ArrFuncs.getitem` (*C member*), 1629
`PyArray_ArrFuncs.nonzero` (*C member*), 1630
`PyArray_ArrFuncs.scalarkind` (*C member*), 1630
`PyArray_ArrFuncs.scanfunc` (*C member*), 1629
`PyArray_ArrFuncs.setitem` (*C member*), 1629
`PyArray_ArrFuncs.sort` (*C member*), 1630
`PyArray_AsCArray` (*C function*), 1671
`PyArray_AxisConverter` (*C function*), 1681
`PyArray_BASE` (*C function*), 1649
`PyArray_BoolConverter` (*C function*), 1681
`PyArray_Broadcast` (*C function*), 1676
`PyArray_BroadcastToShape` (*C function*), 1675
`PyArray_BufferConverter` (*C function*), 1680
`PyArray_ByteorderConverter` (*C function*), 1681
`PyArray_BYTES` (*C function*), 1648
`PyArray_Byteswap` (*C function*), 1666
`PyArray_CanCastArrayTo` (*C function*), 1659

PyArray_CanCastSafely (*C function*), 1659
PyArray_CanCastTo (*C function*), 1659
PyArray_CanCastTypeTo (*C function*), 1659
PyArray_CanCoerceScalar (*C function*), 1678
PyArray_Cast (*C function*), 1659
PyArray_CastingConverter (*C function*), 1681
PyArray_CastScalarToCtype (*C function*), 1678
PyArray_CastTo (*C function*), 1659
PyArray_CastToType (*C function*), 1659
PyArray_CEQ (*C macro*), 1687
PyArray_CGE (*C macro*), 1687
PyArray_CGT (*C macro*), 1687
PyArray_Check (*C function*), 1656
PyArray_CheckAnyScalar (*C function*), 1656
PyArray_CheckAxis (*C function*), 1655
PyArray_CheckExact (*C function*), 1656
PyArray_CheckFromAny (*C function*), 1653
PyArray_CheckScalar (*C function*), 1656
PyArray_CheckStrides (*C function*), 1673
PyArray_CHKFLAGS (*C function*), 1664
PyArray_Choose (*C function*), 1668
PyArray_Chunk (*C type*), 1638
PyArray_Chunk.PyArray_Chunk.base (*C member*), 1638
PyArray_Chunk.PyArray_Chunk.flags (*C member*), 1638
PyArray_Chunk.PyArray_Chunk.len (*C member*), 1638
PyArray_Chunk.PyArray_Chunk.ptr (*C member*), 1638
PyArray_CLE (*C macro*), 1687
PyArray_CLEARFLAGS (*C function*), 1648
PyArray_Clip (*C function*), 1670
PyArray_ClipmodeConverter (*C function*), 1681
PyArray_CLT (*C macro*), 1687
PyArray_CNE (*C macro*), 1687
PyArray_CompareLists (*C function*), 1673
PyArray_Compress (*C function*), 1669
PyArray_Concatenate (*C function*), 1672
PyArray_Conjugate (*C function*), 1670
PyArray_ContiguousFromAny (*C function*), 1654
PyArray_ConvertClipmodeSequence (*C function*), 1681
PyArray_Converter (*C function*), 1680
PyArray_ConvertToCommonType (*C function*), 1660
PyArray_CopyAndTranspose (*C function*), 1672
PyArray_CopyInto (*C function*), 1655
PyArray_Correlate (*C function*), 1672
PyArray_Correlate2 (*C function*), 1673
PyArray_CountNonzero (*C function*), 1669
PyArray_CumProd (*C function*), 1671
PyArray_CumSum (*C function*), 1671
PyArray_DATA (*C function*), 1648
PyArray_DESCR (*C function*), 1649
PyArray_Descr (*C type*), 1625
Pyarray_DescrAlignConverter (*C function*), 1680
Pyarray_DescrAlignConverter2 (*C function*), 1680
PyArray_Descr.alignment (*C member*), 1627
PyArray_Descr.byteorder (*C member*), 1626
PyArray_Descr.c_metadata (*C member*), 1628
PyArray_DescrCheck (*C function*), 1679
PyArray_DescrConverter (*C function*), 1679
PyArray_DescrConverter2 (*C function*), 1679
PyArray_Descr.elsize (*C member*), 1627
PyArray_Descr.f (*C member*), 1628
PyArray_Descr.fields (*C member*), 1628
PyArray_Descr.flags (*C member*), 1626
PyArray_DescrFromObject (*C function*), 1679
PyArray_DescrFromScalar (*C function*), 1679
PyArray_DescrFromType (*C function*), 1679
PyArray_Descr.hash (*C member*), 1628
PyArray_Descr.kind (*C member*), 1626
PyArray_Descr.metadata (*C member*), 1628
PyArray_Descr.names (*C member*), 1628
PyArray_DescrNew (*C function*), 1679
PyArray_DescrNewByteorder (*C function*), 1679
PyArray_DescrNewFromType (*C function*), 1679
PyArray_Descr.subarray (*C member*), 1627
PyArray_Descr.type (*C member*), 1626
PyArray_Descr.type_num (*C member*), 1627
PyArray_Descr.typeobj (*C member*), 1626
PyArray_Diagonal (*C function*), 1669
PyArray_DIM (*C function*), 1648
PyArray_DIMS (*C function*), 1648
PyArray_Dims (*C type*), 1638
PyArray_Dims.PyArray_Dims.len (*C member*), 1638
PyArray_Dims.PyArray_Dims.ptr (*C member*), 1638
PyArray_DiscardWritebackIfCopy (*C function*), 1687
PyArray_DTYPE (*C function*), 1649
PyArray_Dump (*C function*), 1666
PyArray_Dumps (*C function*), 1666
PyArray_EinsteinSum (*C function*), 1672
PyArray_EMPTY (*C function*), 1651
PyArray_Empty (*C function*), 1651
PyArray_ENABLEFLAGS (*C function*), 1648
PyArray_EnsureArray (*C function*), 1654
PyArray_EquivArrTypes (*C function*), 1658
PyArray_EquivByteorders (*C function*), 1658
PyArray_EquivTypenums (*C function*), 1658
PyArray_EquivTypes (*C function*), 1658
PyArray_FieldNames (*C function*), 1680
PyArray_FillObjectArray (*C function*), 1662

PyArray_FILLWBYTE (*C function*), 1651
PyArray_FillWithScalar (*C function*), 1666
PyArray_FLAGS (*C function*), 1648
PyArray_Flatten (*C function*), 1667
PyArray_Free (*C function*), 1672
PyArray_free (*C function*), 1684
PyArray_FROM_O (*C function*), 1655
PyArray_FROM_OF (*C function*), 1655
PyArray_FROM_OT (*C function*), 1655
PyArray_FROM_OTF (*C function*), 1655
PyArray_FROMANY (*C function*), 1655
PyArray_FromAny (*C function*), 1652
PyArray_FromArray (*C function*), 1654
PyArray_FromArrayAttr (*C function*), 1654
PyArray_FromBuffer (*C function*), 1655
PyArray_FromFile (*C function*), 1654
PyArray_FromInterface (*C function*), 1654
PyArray_FromObject (*C function*), 1654
PyArray_FromScalar (*C function*), 1678
PyArray_FromString (*C function*), 1654
PyArray_FromStructInterface (*C function*), 1654
PyArray_GetArrayParamsFromObject (*C function*), 1653
PyArray_GetCastFunc (*C function*), 1659
PyArray_GETCONTIGUOUS (*C function*), 1655
PyArray_GetEndianness (*C function*), 1641
PyArray_GetField (*C function*), 1665
PyArray_GETITEM (*C function*), 1649
PyArray_GetNDArrayCFeatureVersion (*C function*), 1683
PyArray_GetNDArrayCVersion (*C function*), 1683
PyArray_GetNumericOps (*C function*), 1683
PyArray_GetPriority (*C function*), 1686
PyArray_GetPtr (*C function*), 1649
PyArray_GETPTR1 (*C function*), 1649
PyArray_GETPTR2 (*C function*), 1649
PyArray_GETPTR3 (*C function*), 1649
PyArray_GETPTR4 (*C function*), 1649
PyArray_HasArrayInterface (*C function*), 1656
PyArray_HasArrayInterfaceType (*C function*), 1656
PyArray_HASFIELDS (*C function*), 1658
PyArray_INCREF (*C function*), 1662
PyArray_InitArrFuncs (*C function*), 1661
PyArray_InnerProduct (*C function*), 1672
PyArray_IntpConverter (*C function*), 1680
PyArray_IntpFromSequence (*C function*), 1681
PyArray_IS_C_CONTIGUOUS (*C function*), 1664
PyArray_IS_F_CONTIGUOUS (*C function*), 1664
PyArray_ISALIGNED (*C function*), 1665
PyArray_IsAnyScalar (*C function*), 1656
PyArray_ISBEHAVED (*C function*), 1665
PyArray_ISBEHAVED_RO (*C function*), 1665
PyArray_ISBOOL (*C function*), 1658
PyArray_ISBYTESWAPPED (*C function*), 1658
PyArray_ISCARRAY (*C function*), 1665
PyArray_ISCARRAY_RO (*C function*), 1665
PyArray_ISCOMPLEX (*C function*), 1657
PyArray_ISEXTENDED (*C function*), 1658
PyArray_ISFARRAY (*C function*), 1665
PyArray_ISFARRAY_RO (*C function*), 1665
PyArray_ISFLEXIBLE (*C function*), 1657
PyArray_ISFLOAT (*C function*), 1657
PyArray_ISFORTRAN (*C function*), 1665
PyArray_ISINTEGER (*C function*), 1657
PyArray_ISNOTSWAPPED (*C function*), 1658
PyArray_ISNUMBER (*C function*), 1657
PyArray_ISOBJECT (*C function*), 1658
PyArray_ISONESEGMENT (*C function*), 1665
PyArray_ISPYTHON (*C function*), 1657
PyArray_IsPythonNumber (*C function*), 1656
PyArray_IsPythonScalar (*C function*), 1656
PyArray_IsScalar (*C function*), 1656
PyArray_ISSIGNED (*C function*), 1657
PyArray_ISSTRING (*C function*), 1657
PyArray_ISUNSIGNED (*C function*), 1657
PyArray_ISUSERDEF (*C function*), 1658
PyArray_ISWRITEABLE (*C function*), 1665
PyArray_IsZeroDim (*C function*), 1656
PyArray_Item_INCREF (*C function*), 1662
PyArray_Item_XDECREF (*C function*), 1662
PyArray_ITEMSIZE (*C function*), 1648
PyArray_ITER_DATA (*C function*), 1675
PyArray_ITER_GOTO (*C function*), 1675
PyArray_ITER_GOTO1D (*C function*), 1675
PyArray_ITER_NEXT (*C function*), 1675
PyArray_ITER_NOTDONE (*C function*), 1675
PyArray_ITER_RESET (*C function*), 1675
PyArray_IterAllButAxis (*C function*), 1675
PyArray_IterNew (*C function*), 1675
PyArray_LexSort (*C function*), 1668
PyArray_malloc (*C function*), 1684
PyArray_MatrixProduct (*C function*), 1672
PyArray_MatrixProduct2 (*C function*), 1672
PyArray_Max (*C function*), 1670
PyArray_MAX (*C macro*), 1687
PyArray_Mean (*C function*), 1670
PyArray_Min (*C function*), 1670
PyArray_MIN (*C macro*), 1687
PyArray_MinScalarType (*C function*), 1659
PyArray_MoveInto (*C function*), 1655
PyArray_MultiIter_DATA (*C function*), 1676
PyArray_MultiIter_GOTO (*C function*), 1676
PyArray_MultiIter_GOTO1D (*C function*), 1676
PyArray_MultiIter_NEXT (*C function*), 1676
PyArray_MultiIter_NEXTi (*C function*), 1676
PyArray_MultiIter_NOTDONE (*C function*), 1676

PyArray_MultiIter_RESET (*C function*), 1676
PyArray_MultiIterNew (*C function*), 1676
PyArray_MultiplyIntList (*C function*), 1673
PyArray_MultiplyList (*C function*), 1673
PyArray_NBYTES (*C function*), 1649
PyArray_NDIM (*C function*), 1648
PyArray_NeighborhoodIterNew (*C function*), 1676
PyArray_New (*C function*), 1650
PyArray_NewCopy (*C function*), 1666
PyArray_NewFromDescr (*C function*), 1650
PyArray_NewLikeArray (*C function*), 1650
PyArray_Newshape (*C function*), 1667
PyArray_Nonzero (*C function*), 1669
PyArray_ObjectType (*C function*), 1660
PyArray_One (*C function*), 1661
PyArray_OrderConverter (*C function*), 1681
PyArray_OutputConverter (*C function*), 1680
PyArray_Partition (*C function*), 1669
PyArray_Prod (*C function*), 1671
PyArray_PromoteTypes (*C function*), 1660
PyArray_Ptp (*C function*), 1670
PyArray_PutMask (*C function*), 1668
PyArray_PutTo (*C function*), 1668
PyArray_PyIntAsInt (*C function*), 1681
PyArray_PyIntAsIntp (*C function*), 1681
PyArray_Ravel (*C function*), 1667
PyArray_realloc (*C function*), 1684
PyArray_REFCOUNT (*C function*), 1687
PyArray_RegisterCanCast (*C function*), 1661
PyArray_RegisterCastFunc (*C function*), 1661
PyArray_RegisterDataType (*C function*), 1661
PyArray_RemoveSmallest (*C function*), 1676
PyArray_Repeat (*C function*), 1668
PyArray_Reshape (*C function*), 1667
PyArray_Resize (*C function*), 1667
PyArray_ResolveWritebackIfCopy (*C function*), 1684
PyArray_ResultType (*C function*), 1660
PyArray_Return (*C function*), 1678
PyArray_Round (*C function*), 1670
PyArray_SAMESHAPE (*C function*), 1687
PyArray_Scalar (*C function*), 1678
PyArray_ScalarAsCtype (*C function*), 1678
PyArray_ScalarKind (*C function*), 1678
PyArray_SearchsideConverter (*C function*), 1681
PyArray_SearchSorted (*C function*), 1669
PyArray_SetBaseObject (*C function*), 1651
PyArray_SetField (*C function*), 1665
PyArray_SETITEM (*C function*), 1648
PyArray_SetNumericOps (*C function*), 1683
PyArray_SetStringFunction (*C function*), 1684
PyArray_SetUpdateIfCopyBase (*C function*), 1662
PyArray_SetWritebackIfCopyBase (*C function*), 1662
PyArray_SHAPE (*C function*), 1648
PyArray_SimpleNew (*C function*), 1651
PyArray_SimpleNewFromData (*C function*), 1651
PyArray_SimpleNewFromDescr (*C function*), 1651
PyArray_SIZE (*C function*), 1648
PyArray_Size (*C function*), 1648
PyArray_Sort (*C function*), 1668
PyArray_SortkindConverter (*C function*), 1681
PyArray_Squeeze (*C function*), 1667
PyArray_Std (*C function*), 1671
PyArray_STRIDE (*C function*), 1648
PyArray_STRIDES (*C function*), 1648
PyArray_Sum (*C function*), 1671
PyArray_SwapAxes (*C function*), 1667
PyArray_TakeFrom (*C function*), 1668
PyArray_ToFile (*C function*), 1666
PyArray_ToList (*C function*), 1666
PyArray_ToScalar (*C function*), 1678
PyArray_ToString (*C function*), 1666
PyArray_Trace (*C function*), 1670
PyArray_Transpose (*C function*), 1667
PyArray_TYPE (*C function*), 1648
PyArray_Type (*C variable*), 1624
PyArray_TypeObjectFromType (*C function*), 1678
PyArray_TypestrConvert (*C function*), 1681
PyArray_UpdateFlags (*C function*), 1665
PyArray_ValidType (*C function*), 1661
PyArray_View (*C function*), 1666
PyArray_Where (*C function*), 1673
PyArray_XDECREF (*C function*), 1662
PyArray_XDECREF_ERR (*C function*), 1687
PyArray_Zero (*C function*), 1661
PyArray_ZEROS (*C function*), 1651
PyArray_Zeros (*C function*), 1651
PyArrayDescr_Type (*C variable*), 1625
PyArrayFlags_Type (*C variable*), 1637
PyArrayFlagsObject (*C type*), 1637
PyArrayInterface (*C type*), 1639
PyArrayInterface.PyArrayInterface.data (*C member*), 1639
PyArrayInterface.PyArrayInterface.descr (*C member*), 1639
PyArrayInterface.PyArrayInterface.flags (*C member*), 1639
PyArrayInterface.PyArrayInterface.itemsize (*C member*), 1639
PyArrayInterface.PyArrayInterface.nd (*C member*), 1639
PyArrayInterface.PyArrayInterface.shape (*C member*), 1639

PyArrayInterface.PyArrayInterface.strides (*C member*), 1639

PyArrayInterface.PyArrayInterface.two (*C member*), 1639

PyArrayInterface.PyArrayInterface.typekind (*C member*), 1639

PyArrayIter_Check (*C function*), 1675

PyArrayIter_Type (*C variable*), 1634

PyArrayIterObject (*C type*), 1634

PyArrayIterObject.PyArrayIterObject.ao (*C member*), 1635

PyArrayIterObject.PyArrayIterObject.backstrides (*C member*), 1635

PyArrayIterObject.PyArrayIterObject.contiguous (*C member*), 1635

PyArrayIterObject.PyArrayIterObject.coordinates (*C member*), 1635

PyArrayIterObject.PyArrayIterObject.dataptr (*C member*), 1635

PyArrayIterObject.PyArrayIterObject.dims_m1 (*C member*), 1635

PyArrayIterObject.PyArrayIterObject.factors (*C member*), 1635

PyArrayIterObject.PyArrayIterObject.index (*C member*), 1635

PyArrayIterObject.PyArrayIterObject.nd_m1 (*C member*), 1635

PyArrayIterObject.PyArrayIterObject.size (*C member*), 1635

PyArrayIterObject.PyArrayIterObject.strides (*C member*), 1635

PyArrayMapIter_Type (*C variable*), 1640

PyArrayMultiIter_Type (*C variable*), 1636

PyArrayMultiIterObject (*C type*), 1636

PyArrayMultiIterObject.PyArrayMultiIterObject.dimensions (*C member*), 1636

PyArrayMultiIterObject.PyArrayMultiIterObject.index (*C member*), 1636

PyArrayMultiIterObject.PyArrayMultiIterObject.iters (*C member*), 1636

PyArrayMultiIterObject.PyArrayMultiIterObject.nd (*C member*), 1636

PyArrayMultiIterObject.PyArrayMultiIterObject.numiter (*C member*), 1636

PyArrayMultiIterObject.PyArrayMultiIterObject.size (*C member*), 1636

PyArrayNeighborhoodIter_Next (*C function*), 1678

PyArrayNeighborhoodIter_Reset (*C function*), 1677

PyArrayNeighborhoodIter_Type (*C variable*), 1636

PyArrayNeighborhoodIterObject (*C type*), 1636

PyArrayObject (*C type*), 1624

PyArrayObject.base (*C member*), 1625

PyArrayObject.data (*C member*), 1624

PyArrayObject.descr (*C member*), 1625

PyArrayObject.dimensions (*C member*), 1625

PyArrayObject.flags (*C member*), 1625

PyArrayObject.nd (*C member*), 1625

PyArrayObject.PyObject_HEAD (*C macro*), 1624

PyArrayObject.strides (*C member*), 1625

PyArrayObject.weakreflist (*C member*), 1625

PyDataMem_FREE (*C function*), 1684

PyDataMem_NEW (*C function*), 1684

PyDataMem_RENEW (*C function*), 1684

PyDataType_FLAGCHK (*C function*), 1627

PyDataType_HASFIELDS (*C function*), 1658

PyDataType_ISBOOL (*C function*), 1658

PyDataType_ISCOMPLEX (*C function*), 1657

PyDataType_ISEXTENDED (*C function*), 1658

PyDataType_ISFLEXIBLE (*C function*), 1657

PyDataType_ISFLOAT (*C function*), 1657

PyDataType_ISINTEGER (*C function*), 1657

PyDataType_ISNUMBER (*C function*), 1657

PyDataType_ISOBJECT (*C function*), 1658

PyDataType_ISPYTHON (*C function*), 1657

PyDataType_ISSIGNED (*C function*), 1657

PyDataType_ISSTRING (*C function*), 1657

PyDataType_ISUNSIGNED (*C function*), 1657

PyDataType_ISUSERDEF (*C function*), 1658

PyDataType_REFCHK (*C function*), 1627

PyDimMem_FREE (*C function*), 1684

PyDimMem_NEW (*C function*), 1684

PyDimMem_RENEW (*C function*), 1684

Python Enhancement Proposals
  PEP 3118, 419

PyTypeNum_ISBOOL (*C function*), 1658

PyTypeNum_ISCOMPLEX (*C function*), 1657

PyTypeNum_ISEXTENDED (*C function*), 1658

PyTypeNum_ISFLEXIBLE (*C function*), 1657

PyTypeNum_ISFLOAT (*C function*), 1657

PyTypeNum_ISINTEGER (*C function*), 1657

PyTypeNum_ISNUMBER (*C function*), 1657

PyTypeNum_ISOBJECT (*C function*), 1658

PyTypeNum_ISPYTHON (*C function*), 1657

PyTypeNum_ISSIGNED (*C function*), 1657

PyTypeNum_ISSTRING (*C function*), 1657

PyTypeNum_ISUNSIGNED (*C function*), 1657

PyTypeNum_ISUSERDEF (*C function*), 1658

PyUFunc_checkfperr (*C function*), 1709

PyUFunc_clearfperr (*C function*), 1710

PyUFunc_D_D (*C function*), 1710

PyUFunc_d_d (*C function*), 1710

PyUFunc_DD_D (*C function*), 1710

PyUFunc_dd_d (*C function*), 1710

PyUFunc_e_e (*C function*), 1710

PyUFunc_e_e_As_d_d (*C function*), 1710
PyUFunc_e_e_As_f_f (*C function*), 1710
PyUFunc_ee_e (*C function*), 1711
PyUFunc_ee_e_As_dd_d (*C function*), 1711
PyUFunc_ee_e_As_ff_f (*C function*), 1711
PyUFunc_F_F (*C function*), 1710
PyUFunc_f_f (*C function*), 1710
PyUFunc_F_F_As_D_D (*C function*), 1710
PyUFunc_f_f_As_d_d (*C function*), 1710
PyUFunc_FF_F (*C function*), 1710
PyUFunc_ff_f (*C function*), 1710
PyUFunc_FF_F_As_DD_D (*C function*), 1710
PyUFunc_ff_f_As_dd_d (*C function*), 1710
PyUFunc_FromFuncAndData (*C function*), 1707
PyUFunc_FromFuncAndDataAndSignature (*C function*), 1708
PyUFunc_FromFuncAndDataAndSignatureAndIdentity (*C function*), 1709
PyUFunc_G_G (*C function*), 1710
PyUFunc_g_g (*C function*), 1710
PyUFunc_GenericFunction (*C function*), 1709
PyUFunc_GetPyValues (*C function*), 1710
PyUFunc_GG_G (*C function*), 1710
PyUFunc_gg_g (*C function*), 1710
PyUFunc_IdentityValue (*C variable*), 1706
PyUFunc_Loop1d (*C type*), 1640
PyUFunc_MinusOne (*C variable*), 1706
PyUFunc_None (*C variable*), 1706
PyUFunc_O_O (*C function*), 1711
PyUFunc_O_O_method (*C function*), 1711
PyUFunc_On_Om (*C function*), 1711
PyUFunc_One (*C variable*), 1706
PyUFunc_OO_O (*C function*), 1711
PyUFunc_OO_O_method (*C function*), 1711
PyUFunc_PyFuncData (*C type*), 1711
PyUFunc_RegisterLoopForDescr (*C function*), 1709
PyUFunc_RegisterLoopForType (*C function*), 1709
PyUFunc_ReorderableNone (*C variable*), 1706
PyUFunc_ReplaceLoopBySignature (*C function*), 1709
PyUFunc_Type (*C variable*), 1631
PyUFunc_Zero (*C variable*), 1706
PyUFuncLoopObject (*C type*), 1640
PyUFuncObject (*C type*), 1631
PyUFuncObject.PyUFuncObject.core_dim_flags (*C member*), 1634
PyUFuncObject.PyUFuncObject.core_dim_ixs (*C member*), 1633
PyUFuncObject.PyUFuncObject.core_dim_sizes (*C member*), 1634
PyUFuncObject.PyUFuncObject.core_enabled (*C member*), 1633
PyUFuncObject.PyUFuncObject.core_num_dim_ix (*C member*), 1633
PyUFuncObject.PyUFuncObject.core_num_dims (*C member*), 1633
PyUFuncObject.PyUFuncObject.core_offsets (*C member*), 1633
PyUFuncObject.PyUFuncObject.core_signature (*C member*), 1633
PyUFuncObject.PyUFuncObject.data (*C member*), 1633
PyUFuncObject.PyUFuncObject.doc (*C member*), 1633
PyUFuncObject.PyUFuncObject.functions (*C member*), 1632
PyUFuncObject.PyUFuncObject.identity (*C member*), 1632
PyUFuncObject.PyUFuncObject.iter_flags (*C member*), 1634
PyUFuncObject.PyUFuncObject.legacy_inner_loop_selec
(*C member*), 1634
PyUFuncObject.PyUFuncObject.masked_inner_loop_selec
(*C member*), 1634
PyUFuncObject.PyUFuncObject.name (*C member*), 1633
PyUFuncObject.PyUFuncObject.nargs (*C member*), 1632
PyUFuncObject.PyUFuncObject.nin (*C member*), 1632
PyUFuncObject.PyUFuncObject.nout (*C member*), 1632
PyUFuncObject.PyUFuncObject.ntypes (*C member*), 1633
PyUFuncObject.PyUFuncObject.obj (*C member*), 1633
PyUFuncObject.PyUFuncObject.op_flags (*C member*), 1634
PyUFuncObject.PyUFuncObject.ptr (*C member*), 1633
PyUFuncObject.PyUFuncObject.reserved1 (*C member*), 1633
PyUFuncObject.PyUFuncObject.reserved2 (*C member*), 1634
PyUFuncObject.PyUFuncObject.type_resolver (*C member*), 1634
PyUFuncObject.PyUFuncObject.types (*C member*), 1633
PyUFuncObject.PyUFuncObject.userloops (*C member*), 1633
PyUFuncReduceObject (*C type*), 1640
PZERO (*in module numpy*), 433

Q

qr() (*in module numpy.linalg*), 817
quantile() (*in module numpy*), 1518

quote_args() (*in module numpy.distutils.misc_util*), 1596

## R

r_ (*in module numpy*), 717
rad2deg (*in module numpy*), 888
radians (*in module numpy*), 885
rand() (*in module numpy.matlib*), 990
rand() (*in module numpy.random*), 1410
rand() (*numpy.random.RandomState method*), 1296
randint() (*in module numpy.random*), 1411
randint() (*numpy.random.RandomState method*), 1298
randn() (*in module numpy.matlib*), 991
randn() (*in module numpy.random*), 1413
randn() (*numpy.random.RandomState method*), 1297
random() (*in module numpy.random*), 1414
random() (*numpy.random.Generator method*), 1230
random_beta (*C function*), 1463
random_binomial (*C function*), 1464
random_bounded_uint64 (*C function*), 1465
random_chisquare (*C function*), 1463
random_exponential (*C function*), 1463
random_f (*C function*), 1463
random_gamma (*C function*), 1463
random_gamma_f (*C function*), 1463
random_geometric (*C function*), 1464
random_geometric_inversion (*C function*), 1464
random_geometric_search (*C function*), 1464
random_gumbel (*C function*), 1463
random_hypergeometric (*C function*), 1464
random_integers() (*in module numpy.random*), 1414
random_integers() (*numpy.random.RandomState method*), 1299
random_interval (*C function*), 1464
random_laplace (*C function*), 1463
random_logistic (*C function*), 1463
random_lognormal (*C function*), 1463
random_logseries (*C function*), 1464
random_multinomial (*C function*), 1464
random_multivariate_hypergeometric_count (*C function*), 1464
random_multivariate_hypergeometric_marginals (*C function*), 1464
random_negative_binomial (*C function*), 1464
random_noncentral_chisquare (*C function*), 1463
random_noncentral_f (*C function*), 1463
random_normal (*C function*), 1463
random_pareto (*C function*), 1463
random_poisson (*C function*), 1464
random_positive_int (*C function*), 1465
random_positive_int32 (*C function*), 1465
random_positive_int64 (*C function*), 1464
random_power (*C function*), 1463
random_raw() (*numpy.random.BitGenerator method*), 1438
random_rayleigh (*C function*), 1463
random_sample() (*in module numpy.random*), 1415
random_sample() (*numpy.random.RandomState method*), 1301
random_standard_cauchy (*C function*), 1463
random_standard_exponential (*C function*), 1463
random_standard_exponential_f (*C function*), 1463
random_standard_exponential_fill (*C function*), 1463
random_standard_exponential_fill_f (*C function*), 1463
random_standard_gamma (*C function*), 1463
random_standard_gamma_f (*C function*), 1463
random_standard_normal (*C function*), 1463
random_standard_normal_f (*C function*), 1463
random_standard_normal_fill (*C function*), 1463
random_standard_normal_fill_f (*C function*), 1463
random_standard_t (*C function*), 1463
random_standard_uniform (*C function*), 1463
random_standard_uniform_f (*C function*), 1463
random_standard_uniform_fill (*C function*), 1463
random_standard_uniform_fill_f (*C function*), 1463
random_triangular (*C function*), 1464
random_uint (*C function*), 1465
random_uniform (*C function*), 1463
random_vonmises (*C function*), 1464
random_wald (*C function*), 1463
random_weibull (*C function*), 1463
random_zipf (*C function*), 1464
RandomState (*class in numpy.random*), 1293
ranf() (*in module numpy.random*), 1416
RankWarning, 1203, 1224
rate() (*in module numpy*), 704
ravel (*in module numpy.ma*), 337
ravel() (*in module numpy*), 516
ravel() (*numpy.chararray method*), 177
ravel() (*numpy.char.chararray method*), 616
ravel() (*numpy.generic method*), 69
ravel() (*numpy.ma.MaskedArray method*), 278
ravel() (*numpy.matrix method*), 143
ravel() (*numpy.ndarray method*), 22
ravel() (*numpy.recarray method*), 209
ravel() (*numpy.record method*), 230
ravel_multi_index() (*in module numpy*), 724
rayleigh() (*in module numpy.random*), 1417

`rayleigh()` (*numpy.random.Generator method*), 1274
`rayleigh()` (*numpy.random.RandomState method*), 1346
`real` (*numpy.generic attribute*), 61
`real` (*numpy.ndarray attribute*), 43
`real()` (*in module numpy*), 961
`real()` (*numpy.ma.MaskedArray property*), 268
`real_if_close()` (*in module numpy*), 982
`recarray` (*class in numpy*), 191
`reciprocal` (*in module numpy*), 944
`record` (*class in numpy*), 223
`record()` (*numpy.testing.suppress_warnings method*), 1573
`recordmask` (*numpy.ma.MaskedArray attribute*), 259
`red_text()` (*in module numpy.distutils.misc_util*), 1595
`reduce`
    `ufunc` methods, 1729
`reduce()` (*numpy.ufunc method*), 452
`reduceat`
    `ufunc` methods, 1730
`reduceat()` (*numpy.ufunc method*), 455
`remainder` (*in module numpy*), 957
`remove_axis()` (*numpy.nditer method*), 756
`remove_multi_index()` (*numpy.nditer method*), 757
`repeat()` (*in module numpy*), 549
`repeat()` (*numpy.chararray method*), 177
`repeat()` (*numpy.char.chararray method*), 616
`repeat()` (*numpy.generic method*), 69
`repeat()` (*numpy.ma.MaskedArray method*), 291
`repeat()` (*numpy.matrix method*), 144
`repeat()` (*numpy.ndarray method*), 22
`repeat()` (*numpy.recarray method*), 210
`repeat()` (*numpy.record method*), 230
`replace()` (*in module numpy.char*), 583
`replace()` (*numpy.chararray method*), 177
`replace()` (*numpy.char.chararray method*), 616
`repmat()` (*in module numpy.matlib*), 989
`require()` (*in module numpy*), 531
`reset()` (*numpy.broadcast method*), 237
`reset()` (*numpy.nditer method*), 757
`reshape()` (*in module numpy*), 514
`reshape()` (*in module numpy.ma*), 338
`reshape()` (*numpy.chararray method*), 177
`reshape()` (*numpy.char.chararray method*), 616
`reshape()` (*numpy.generic method*), 69
`reshape()` (*numpy.ma.MaskedArray method*), 278
`reshape()` (*numpy.matrix method*), 144
`reshape()` (*numpy.ndarray method*), 22
`reshape()` (*numpy.recarray method*), 210
`reshape()` (*numpy.record method*), 230
`resize()` (*in module numpy*), 554
`resize()` (*in module numpy.ma*), 338
`resize()` (*numpy.chararray method*), 178
`resize()` (*numpy.char.chararray method*), 617
`resize()` (*numpy.generic method*), 70
`resize()` (*numpy.ma.MaskedArray method*), 280
`resize()` (*numpy.matrix method*), 144
`resize()` (*numpy.ndarray method*), 22
`resize()` (*numpy.recarray method*), 210
`resize()` (*numpy.record method*), 230
`result_type()` (*in module numpy*), 644
`rfft()` (*in module numpy.fft*), 677
`rfft2()` (*in module numpy.fft*), 680
`rfftfreq()` (*in module numpy.fft*), 689
`rfftn()` (*in module numpy.fft*), 682
`rfind()` (*in module numpy.char*), 599
`rfind()` (*numpy.chararray method*), 179
`rfind()` (*numpy.char.chararray method*), 618
`right_shift` (*in module numpy*), 570
`rindex()` (*in module numpy.char*), 599
`rindex()` (*numpy.chararray method*), 179
`rindex()` (*numpy.char.chararray method*), 618
`rint` (*in module numpy*), 898
`rjust()` (*in module numpy.char*), 583
`rjust()` (*numpy.chararray method*), 179
`rjust()` (*numpy.char.chararray method*), 618
`roll()` (*in module numpy*), 561
`rollaxis()` (*in module numpy*), 519
`roots()` (*in module numpy*), 1214
`roots()` (*numpy.polynomial.chebyshev.Chebyshev method*), 1051
`roots()` (*numpy.polynomial.hermite_e.HermiteE method*), 1116
`roots()` (*numpy.polynomial.hermite.Hermite method*), 1085
`roots()` (*numpy.polynomial.laguerre.Laguerre method*), 1148
`roots()` (*numpy.polynomial.legendre.Legendre method*), 1179
`roots()` (*numpy.polynomial.polynomial.Polynomial method*), 1019
`rot90()` (*in module numpy*), 562
`round()` (*in module numpy.ma*), 408
`round()` (*numpy.generic method*), 70
`round()` (*numpy.ma.MaskedArray method*), 302
`round()` (*numpy.matrix method*), 146
`round()` (*numpy.ndarray method*), 24
`round()` (*numpy.recarray method*), 212
`round()` (*numpy.record method*), 230
`round_()` (*in module numpy*), 898
`row_stack` (*in module numpy.ma*), 352
`row-major`, 35
`rpartition()` (*in module numpy.char*), 584
`rsplit()` (*in module numpy.char*), 584
`rsplit()` (*numpy.chararray method*), 180
`rsplit()` (*numpy.char.chararray method*), 619
`rstrip()` (*in module numpy.char*), 584
`rstrip()` (*numpy.chararray method*), 180

`rstrip()` (*numpy.char.chararray method*), 619
`run_module_suite()` (*in module numpy.testing*), 1570
`rundocs()` (*in module numpy.testing*), 1570

## S

`s_` (*in module numpy*), 718
`sample()` (*in module numpy.random*), 1418
`save()` (*in module numpy*), 763
`savetxt()` (*in module numpy*), 767
`savez()` (*in module numpy*), 765
`savez_compressed()` (*in module numpy*), 766
`scalar`
    `dtype`, 73
`sctype2char()` (*in module numpy*), 656
`searchsorted()` (*in module numpy*), 1501
`searchsorted()` (*numpy.chararray method*), 180
`searchsorted()` (*numpy.char.chararray method*), 619
`searchsorted()` (*numpy.generic method*), 70
`searchsorted()` (*numpy.ma.MaskedArray method*), 291
`searchsorted()` (*numpy.matrix method*), 146
`searchsorted()` (*numpy.ndarray method*), 24
`searchsorted()` (*numpy.recarray method*), 212
`searchsorted()` (*numpy.record method*), 230
`seed()` (*in module numpy.random*), 1418
`seed()` (*numpy.random.RandomState method*), 1295
`SeedSequence` (*class in numpy.random*), 1452
`select()` (*in module numpy*), 742
`set_fill_value()` (*in module numpy.ma*), 374
`set_fill_value()` (*numpy.ma.MaskedArray method*), 315
`set_printoptions()` (*in module numpy*), 782
`set_state()` (*in module numpy.random*), 1418
`set_state()` (*numpy.random.RandomState method*), 1295
`set_string_function()` (*in module numpy*), 785
`set_verbosity()` (*in module numpy.distutils.log*), 1601
`setastest()` (*in module numpy.testing.dec*), 1568
`setbufsize()` (*in module numpy*), 440
`setdiff1d()` (*in module numpy*), 1482
`seterr()` (*in module numpy*), 441
`seterrcall()` (*in module numpy*), 442
`seterrobj()` (*in module numpy*), 663
`setfield()` (*numpy.chararray method*), 180
`setfield()` (*numpy.char.chararray method*), 619
`setfield()` (*numpy.generic method*), 70
`setfield()` (*numpy.matrix method*), 146
`setfield()` (*numpy.ndarray method*), 24
`setfield()` (*numpy.recarray method*), 212
`setfield()` (*numpy.record method*), 230
`setflags()` (*numpy.chararray method*), 181
`setflags()` (*numpy.char.chararray method*), 620
`setflags()` (*numpy.generic method*), 70
`setflags()` (*numpy.matrix method*), 147
`setflags()` (*numpy.ndarray method*), 25
`setflags()` (*numpy.recarray method*), 213
`setflags()` (*numpy.record method*), 231
`setitem`
    `ndarray special methods`, 92
`setxor1d()` (*in module numpy*), 1483
`SFC64` (*class in numpy.random*), 1449
`shape` (*numpy.dtype attribute*), 88
`shape` (*numpy.generic attribute*), 60
`shape` (*numpy.ndarray attribute*), 38
`shape()` (*in module numpy*), 513
`shape()` (*in module numpy.ma*), 334
`shape()` (*numpy.ma.MaskedArray property*), 265
`sharedmask` (*numpy.ma.MaskedArray attribute*), 260
`shares_memory()` (*in module numpy*), 992
`show_config()` (*in module numpy*), 997
`shrink_mask()` (*numpy.ma.MaskedArray method*), 314
`shuffle()` (*in module numpy.random*), 1419
`shuffle()` (*numpy.random.Generator method*), 1233
`shuffle()` (*numpy.random.RandomState method*), 1304
`sign` (*in module numpy*), 972
`signature` (*numpy.ufunc attribute*), 451
`signbit` (*in module numpy*), 935
`sin` (*in module numpy*), 873
`sinc()` (*in module numpy*), 934
`single-segment`, 35
`sinh` (*in module numpy*), 889
`size` (*numpy.generic attribute*), 61
`size` (*numpy.ma.MaskedArray attribute*), 266
`size` (*numpy.ndarray attribute*), 40
`size()` (*in module numpy.ma*), 334
`skipif()` (*in module numpy.testing.dec*), 1568
`slicing`, 91
`slogdet()` (*in module numpy.linalg*), 834
`slow()` (*in module numpy.testing.dec*), 1569
`soften_mask` (*in module numpy.ma*), 368
`soften_mask()` (*numpy.ma.MaskedArray method*), 314
`solve()` (*in module numpy.linalg*), 837
`sort()` (*in module numpy*), 1485
`sort()` (*in module numpy.ma*), 393
`sort()` (*numpy.chararray method*), 182
`sort()` (*numpy.char.chararray method*), 621
`sort()` (*numpy.generic method*), 70
`sort()` (*numpy.ma.MaskedArray method*), 291
`sort()` (*numpy.matrix method*), 148
`sort()` (*numpy.ndarray method*), 26
`sort()` (*numpy.recarray method*), 214
`sort()` (*numpy.record method*), 231
`sort_complex()` (*in module numpy*), 1491
`source()` (*in module numpy*), 715

`spacing` (*in module numpy*), 940
`spawn()` (*numpy.random.SeedSequence method*), 1453
`special methods`
    `getitem, ndarray`, 92
    `setitem, ndarray`, 92
`split()` (*in module numpy*), 543
`split()` (*in module numpy.char*), 585
`split()` (*numpy.chararray method*), 183
`split()` (*numpy.char.chararray method*), 622
`splitlines()` (*in module numpy.char*), 586
`splitlines()` (*numpy.chararray method*), 184
`splitlines()` (*numpy.char.chararray method*), 623
`sqrt` (*in module numpy*), 967
`square` (*in module numpy*), 969
`squeeze()` (*in module numpy*), 527
`squeeze()` (*in module numpy.ma*), 344
`squeeze()` (*numpy.chararray method*), 184
`squeeze()` (*numpy.char.chararray method*), 623
`squeeze()` (*numpy.generic method*), 71
`squeeze()` (*numpy.ma.MaskedArray method*), 280
`squeeze()` (*numpy.matrix method*), 150
`squeeze()` (*numpy.ndarray method*), 28
`squeeze()` (*numpy.recarray method*), 216
`squeeze()` (*numpy.record method*), 231
`stack` (*in module numpy.ma*), 345
`stack()` (*in module numpy*), 535
`standard_cauchy()` (*in module numpy.random*), 1420
`standard_cauchy()` (*numpy.random.Generator method*), 1276
`standard_cauchy()` (*numpy.random.RandomState method*), 1349
`standard_exponential()` (*in module numpy.random*), 1421
`standard_exponential()` (*numpy.random.Generator method*), 1277
`standard_exponential()` (*numpy.random.RandomState method*), 1350
`standard_gamma()` (*in module numpy.random*), 1422
`standard_gamma()` (*numpy.random.Generator method*), 1278
`standard_gamma()` (*numpy.random.RandomState method*), 1351
`standard_normal()` (*in module numpy.random*), 1423
`standard_normal()` (*numpy.random.Generator method*), 1279
`standard_normal()` (*numpy.random.RandomState method*), 1352
`standard_t()` (*in module numpy.random*), 1425
`standard_t()` (*numpy.random.Generator method*), 1281
`standard_t()` (*numpy.random.RandomState method*), 1354
`startswith()` (*in module numpy.char*), 600
`startswith()` (*numpy.chararray method*), 184
`startswith()` (*numpy.char.chararray method*), 623
`state` (*numpy.random.MT19937 attribute*), 1440
`state` (*numpy.random.PCG64 attribute*), 1443
`state` (*numpy.random.Philox attribute*), 1447
`state` (*numpy.random.SFC64 attribute*), 1450
`std` (*in module numpy.ma*), 385
`std()` (*in module numpy*), 1527
`std()` (*numpy.generic method*), 71
`std()` (*numpy.ma.MaskedArray method*), 302
`std()` (*numpy.matrix method*), 151
`std()` (*numpy.ndarray method*), 28
`std()` (*numpy.recarray method*), 216
`std()` (*numpy.record method*), 231
`str` (*numpy.dtype attribute*), 85
`str_len()` (*in module numpy.char*), 600
`stride`, 35
`strides` (*numpy.generic attribute*), 61
`strides` (*numpy.ma.MaskedArray attribute*), 266
`strides` (*numpy.ndarray attribute*), 39
`strip()` (*in module numpy.char*), 586
`strip()` (*numpy.chararray method*), 184
`strip()` (*numpy.char.chararray method*), 623
`sub-array`
    `dtype`, 73, 81
`subdtype` (*numpy.dtype attribute*), 88
`subtract` (*in module numpy*), 949
`sum` (*in module numpy.ma*), 385
`sum()` (*in module numpy*), 904
`sum()` (*numpy.generic method*), 71
`sum()` (*numpy.ma.MaskedArray method*), 302
`sum()` (*numpy.matrix method*), 151
`sum()` (*numpy.ndarray method*), 28
`sum()` (*numpy.recarray method*), 216
`sum()` (*numpy.record method*), 231
`suppress_warnings` (*class in numpy.testing*), 1571
`svd()` (*in module numpy.linalg*), 819
`swapaxes` (*in module numpy.ma*), 339
`swapaxes()` (*in module numpy*), 520
`swapaxes()` (*numpy.chararray method*), 184
`swapaxes()` (*numpy.char.chararray method*), 623
`swapaxes()` (*numpy.generic method*), 71
`swapaxes()` (*numpy.ma.MaskedArray method*), 280
`swapaxes()` (*numpy.matrix method*), 152
`swapaxes()` (*numpy.ndarray method*), 28
`swapaxes()` (*numpy.recarray method*), 216
`swapaxes()` (*numpy.record method*), 231
`swapcase()` (*in module numpy.char*), 587
`swapcase()` (*numpy.chararray method*), 184
`swapcase()` (*numpy.char.chararray method*), 623

## T

`T` (*numpy.generic attribute*), 61

T (*numpy.ndarray attribute*), 42
T() (*numpy.ma.MaskedArray property*), 281
T() (*numpy.matrix property*), 117
take() (*in module numpy*), 733
take() (*numpy.chararray method*), 184
take() (*numpy.char.chararray method*), 623
take() (*numpy.generic method*), 71
take() (*numpy.ma.MaskedArray method*), 293
take() (*numpy.matrix method*), 152
take() (*numpy.ndarray method*), 28
take() (*numpy.recarray method*), 216
take() (*numpy.record method*), 231
take_along_axis() (*in module numpy*), 735
tan (*in module numpy*), 876
tanh (*in module numpy*), 891
tensordot() (*in module numpy*), 803
tensorinv() (*in module numpy.linalg*), 844
tensorsolve() (*in module numpy.linalg*), 838
terminal_has_colors() (*in module numpy.distutils.misc_util*), 1595
Tester (*in module numpy.testing*), 1570
tile() (*in module numpy*), 548
title() (*in module numpy.char*), 587
title() (*numpy.chararray method*), 185
title() (*numpy.char.chararray method*), 624
tobytes() (*numpy.ma.MaskedArray method*), 276
tobytes() (*numpy.matrix method*), 152
tobytes() (*numpy.ndarray method*), 29
tobytes() (*numpy.recarray method*), 217
todict() (*numpy.distutils.misc_util.Configuration method*), 1601
tofile() (*numpy.chararray method*), 185
tofile() (*numpy.char.chararray method*), 624
tofile() (*numpy.generic method*), 71
tofile() (*numpy.ma.MaskedArray method*), 273
tofile() (*numpy.matrix method*), 153
tofile() (*numpy.ndarray method*), 29
tofile() (*numpy.recarray method*), 217
tofile() (*numpy.record method*), 232
toflex() (*numpy.ma.MaskedArray method*), 274
tolist() (*numpy.chararray method*), 185
tolist() (*numpy.char.chararray method*), 624
tolist() (*numpy.generic method*), 71
tolist() (*numpy.ma.MaskedArray method*), 274
tolist() (*numpy.matrix method*), 153
tolist() (*numpy.ndarray method*), 30
tolist() (*numpy.recarray method*), 218
tolist() (*numpy.record method*), 232
torecords() (*numpy.ma.MaskedArray method*), 275
tostring() (*numpy.chararray method*), 187
tostring() (*numpy.char.chararray method*), 626
tostring() (*numpy.generic method*), 72
tostring() (*numpy.ma.MaskedArray method*), 276
tostring() (*numpy.matrix method*), 154
tostring() (*numpy.ndarray method*), 31
tostring() (*numpy.recarray method*), 219
tostring() (*numpy.record method*), 232
trace (*in module numpy.ma*), 402
trace() (*in module numpy*), 836
trace() (*numpy.generic method*), 72
trace() (*numpy.ma.MaskedArray method*), 303
trace() (*numpy.matrix method*), 154
trace() (*numpy.ndarray method*), 31
trace() (*numpy.recarray method*), 219
trace() (*numpy.record method*), 232
translate() (*in module numpy.char*), 588
translate() (*numpy.chararray method*), 187
translate() (*numpy.char.chararray method*), 626
transpose() (*in module numpy*), 520
transpose() (*in module numpy.ma*), 340
transpose() (*numpy.chararray method*), 187
transpose() (*numpy.char.chararray method*), 626
transpose() (*numpy.generic method*), 72
transpose() (*numpy.ma.MaskedArray method*), 280
transpose() (*numpy.matrix method*), 154
transpose() (*numpy.ndarray method*), 31
transpose() (*numpy.recarray method*), 219
transpose() (*numpy.record method*), 232
trapz() (*in module numpy*), 921
tri() (*in module numpy*), 509
triangular() (*in module numpy.random*), 1426
triangular() (*numpy.random.Generator method*), 1282
triangular() (*numpy.random.RandomState method*), 1355
tril() (*in module numpy*), 509
tril_indices() (*in module numpy*), 729
tril_indices_from() (*in module numpy*), 730
trim() (*numpy.polynomial.chebyshev.Chebyshev method*), 1051
trim() (*numpy.polynomial.hermite_e.HermiteE method*), 1117
trim() (*numpy.polynomial.hermite.Hermite method*), 1085
trim() (*numpy.polynomial.laguerre.Laguerre method*), 1148
trim() (*numpy.polynomial.legendre.Legendre method*), 1180
trim() (*numpy.polynomial.polynomial.Polynomial method*), 1020
trim_zeros() (*in module numpy*), 555
trimcoef() (*in module numpy.polynomial.polyutils*), 1204
trimseq() (*in module numpy.polynomial.polyutils*), 1204
triu() (*in module numpy*), 510
triu_indices() (*in module numpy*), 731
triu_indices_from() (*in module numpy*), 732

true_divide (*in module numpy*), 950
trunc (*in module numpy*), 901
truncate() (*numpy.polynomial.chebyshev.Chebyshev method*), 1051
truncate() (*numpy.polynomial.hermite_e.HermiteE method*), 1117
truncate() (*numpy.polynomial.hermite.Hermite method*), 1085
truncate() (*numpy.polynomial.laguerre.Laguerre method*), 1148
truncate() (*numpy.polynomial.legendre.Legendre method*), 1180
truncate() (*numpy.polynomial.polynomial.Polynomial method*), 1020
type (*numpy.dtype attribute*), 84
typename() (*in module numpy*), 655
types (*numpy.ufunc attribute*), 450

## U

ufunc, 1726, 1730
  attributes, 448
  C-API, 1706, 1712
  casting rules, 444
  keyword arguments, 446
  methods, 452
  methods accumulate, 1729
  methods reduce, 1729
  methods reduceat, 1730
uniform() (*in module numpy.random*), 1428
uniform() (*numpy.random.Generator method*), 1284
uniform() (*numpy.random.RandomState method*), 1357
union1d() (*in module numpy*), 1484
unique() (*in module numpy*), 555
unpackbits() (*in module numpy*), 572
unravel_index() (*in module numpy*), 725
unshare_mask() (*numpy.ma.MaskedArray method*), 314
unwrap() (*in module numpy*), 886
upper() (*in module numpy.char*), 588
upper() (*numpy.chararray method*), 188
upper() (*numpy.char.chararray method*), 627
user_array, 233

## V

vander() (*in module numpy*), 510
vander() (*in module numpy.ma*), 402
var (*in module numpy.ma*), 386
var() (*in module numpy*), 1529
var() (*numpy.generic method*), 72
var() (*numpy.ma.MaskedArray method*), 303
var() (*numpy.matrix method*), 155
var() (*numpy.ndarray method*), 32
var() (*numpy.recarray method*), 220
var() (*numpy.record method*), 232
vdot() (*in module numpy*), 797
vectorize (*class in numpy*), 708
view, 3
  ndarray, 94
view() (*numpy.chararray method*), 188
view() (*numpy.char.chararray method*), 627
view() (*numpy.generic method*), 72
view() (*numpy.ma.MaskedArray method*), 269
view() (*numpy.matrix method*), 156
view() (*numpy.ndarray method*), 33
view() (*numpy.recarray method*), 221
view() (*numpy.record method*), 233
vonmises() (*in module numpy.random*), 1430
vonmises() (*numpy.random.Generator method*), 1285
vonmises() (*numpy.random.RandomState method*), 1359
vsplit() (*in module numpy*), 547
vstack (*in module numpy.ma*), 353
vstack() (*in module numpy*), 539

## W

wald() (*in module numpy.random*), 1432
wald() (*numpy.random.Generator method*), 1287
wald() (*numpy.random.RandomState method*), 1361
weibull() (*in module numpy.random*), 1433
weibull() (*numpy.random.Generator method*), 1288
weibull() (*numpy.random.RandomState method*), 1362
where() (*in module numpy*), 720
where() (*in module numpy.ma*), 418
who() (*in module numpy*), 999

## Y

yellow_text() (*in module numpy.distutils.misc_util*), 1596

## Z

zeros (*in module numpy.ma*), 325
zeros() (*in module numpy*), 471
zeros() (*in module numpy.matlib*), 986
zeros_like() (*in module numpy*), 472
zfill() (*in module numpy.char*), 589
zfill() (*numpy.chararray method*), 190
zfill() (*numpy.char.chararray method*), 629
zipf() (*in module numpy.random*), 1435
zipf() (*numpy.random.Generator method*), 1290
zipf() (*numpy.random.RandomState method*), 1364