# NumPy: Array Manipulation

Hendrik Speleers

# NumPy: Array Manipulation

Università di Roma Tor Vergata

- Overview
  - 1D and 2D arrays
    - Creation, indexing and slicing
    - Memory structure
  - Shape manipulation
  - Basic mathematical operations
    - Arithmetic and logic operations
    - Reduction and linear algebra operations
  - Other operations
    - Polynomial manipulation
    - Input and output

# NumPy: Array Manipulation

- NumPy
  - Numerical Python
  - Python extension for multi-dimensional arrays
    - Suited for creation and manipulation of numerical data
    - Closer to hardware: more efficient
    - Designed for scientific computation: more intuitive
  - Import convention

```
import numpy as np
```

NumPy

# NumPy: Array Manipulation



- NumPy array
  - A NumPy array is a collection of objects of the same type

```
In [1]: a = np.array([0, 1, 2, 3])
In [2]: a
Out[2]: array([0, 1, 2, 3])
In [3]: a.size
Out[3]: 4
```

  - Default object types of an array
    - boolean (`bool`), integer (`int`, `int64`)
    - float (`float`, `float64`), complex (`complex`, `complex128`)

# NumPy: Array Manipulation

- NumPy array
  - More compact and more efficient operations than list

```
In [1]: L = 100000
In [2]: a = range(L)
In [3]: %timeit [i**2 for i in a]
16.4 ms ± 8.6 µs per loop (mean ± std. dev. of 7
runs, 100 loops each)
In [4]: b = np.arange(L)
In [5]: %timeit b**2
33.7 µs ± 43.4 ns per loop (mean ± std. dev. of 7
runs, 10000 loops each)
```

# NumPy: Array Manipulation

- 1D array: creation
  - Manual creation

```
In [1]: a = np.array([1, 2, 3])
   ...: a.dtype
Out[1]: dtype('int64')
In [2]: a = np.array([1.0, 2.0, 3.0])
   ...: a.dtype
Out[2]: dtype('float64')
In [3]: a = np.array([1, 2, 3], dtype='float64')
   ...: a.dtype
Out[3]: dtype('float64')
```

# NumPy: Array Manipulation



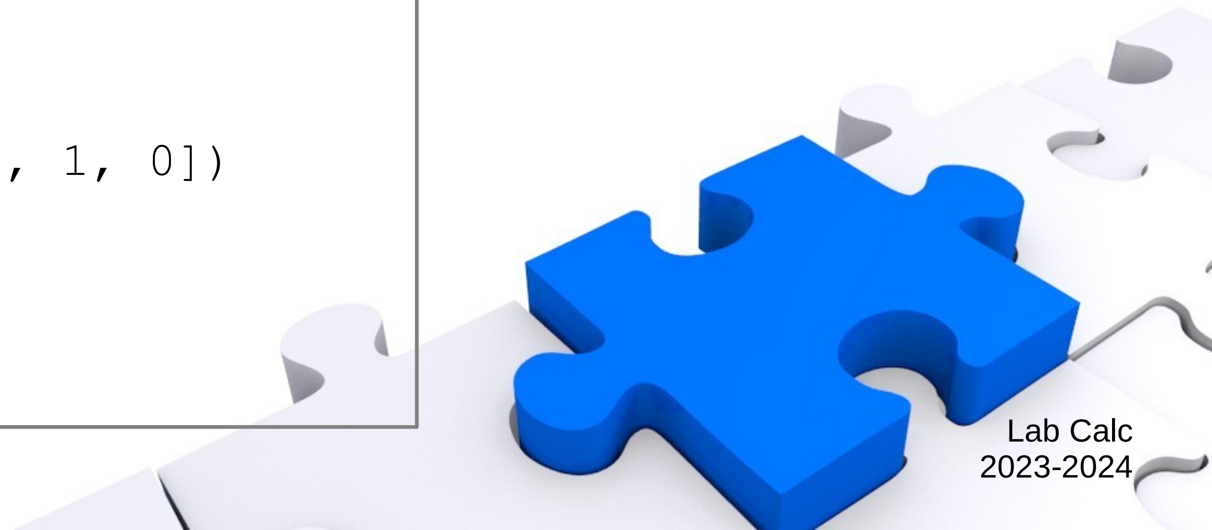
- 1D array: creation
  - Evenly spaced arrays
    - `np.arange(start, stop, step, dtype=None)`
    - `np.linspace(start, stop, num=50, endpoint=True, dtype=None)`
  - Common arrays
    - `np.zeros(N, dtype=None)`, `np.ones(N, dtype=None)`
    - `np.full(N, value, dtype=None)`
  - Arrays with random numbers
    - Uniform distribution: `np.random.rand(N)`
    - Gaussian distribution: `np.random.randn(N)`

# NumPy: Array Manipulation

Università di Roma Tor Vergata

- 1D array: indexing
  - Slicing syntax similar to lists

```
In [1]: a = np.arange(10)
In [2]: a[0], a[1], a[-1]
Out[2]: (0, 1, 9)
In [3]: a[3:6]
Out[3]: array([3, 4, 5])
In [4]: a[::-1]
Out[4]: array([9, 8, 7, 6, 5, 4, 3, 2, 1, 0])
In [5]: b = a[6:8]
In [6]: b
Out[6]: array([6, 7])
```



# NumPy: Array Manipulation



- 1D array: indexing
  - Slicing syntax similar to lists

```
In [1]: a = np.arange(10)
In [2]: a[3] = 1
In [3]: a[-1] = 0
In [4]: a[6:8] = np.array([2, 0])
In [5]: a
Out[5]: array([0, 1, 2, 1, 4, 5, 2, 0, 8, 0])
In [6]: a[6:] = 10
In [7]: a
Out[7]: array([0, 1, 2, 1, 4, 5, 10, 10, 10, 10])
```

Not allowed for list!

# NumPy: Array Manipulation

Università di Roma Tor Vergata

- 1D array: indexing
  - A slicing operation creates a view, not a copy (memory efficiency)

```
In [1]: a = np.arange(10)
In [2]: b = a[::2]
In [3]: b[0] = 11
In [4]: a
Out[4]: array([11, 1, 2, 3, 4, 5, 6, 7, 8, 9])
In [5]: c = a[::2].copy()
In [6]: c[0] = 99
In [7]: a
Out[7]: array([11, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Be careful:
differs from list!

# NumPy: Array Manipulation

- 1D array: indexing
  - Fancy indexing: boolean masks or integer lists

```
In [1]: a = np.arange(10)
In [2]: a[a > 5]
Out[2]: array([6, 7, 8, 9])
In [3]: a[[2, 3, 2, 4, 2]]
Out[3]: array([2, 3, 2, 4, 2])
In [4]: a[[9, 7]] = -9
In [5]: a[a > 0] = 1
In [6]: a
Out[6]: array([0, 1, 1, 1, 1, 1, 1, -9, 1, -9])
```

Fancy indexing not supported for list!

# NumPy: Array Manipulation

Università di Roma Tor Vergata

- 1D array: indexing
  - Fancy indexing creates a copy, not a view

```
In [1]: a = np.arange(10)
In [2]: b = a[a > 5]
In [3]: b[0] = 11
In [4]: a
Out[4]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
In [5]: c = a[[2, 3, 2, 4, 2]]
In [6]: c[0] = 99
In [7]: a
Out[7]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

# NumPy: Array Manipulation

- 1D array: indexing
  - The object type of an array is fixed

```
In [1]: a = np.ones(5, dtype='int64')
In [2]: a
Out[2]: array([1, 1, 1, 1, 1])
In [3]: a[[0, 1]] = [0, 3.5]
In [4]: a
Out[4]: array([0, 3, 1, 1, 1])
In [5]: b = a.astype('float64')
In [6]: b[0] = 3.5
In [7]: b
Out[7]: array([3.5, 3., 1., 1., 1.])
```

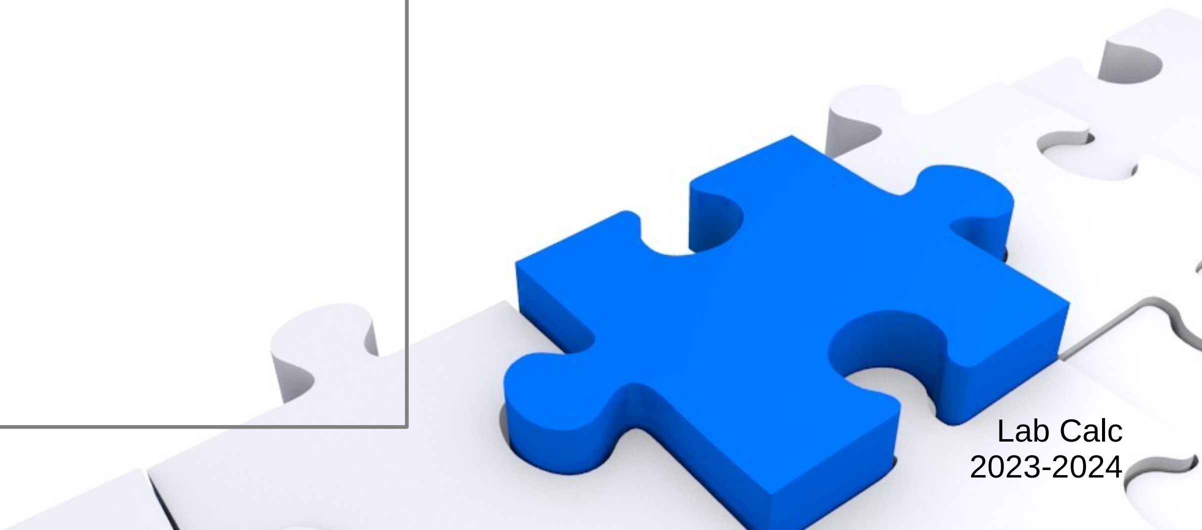
Be careful:
differs from list!

# NumPy: Array Manipulation

Università di Roma Tor Vergata

- 2D array: creation
  - Manual construction

```
In [1]: a = np.array([[0, 1, 2], [3, 4, 5]])
In [2]: a
Out[2]: array([[0, 1, 2],
   ...:        [3, 4, 5]])
In [3]: a.ndim
Out[3]: 2
In [4]: a.shape
Out[4]: (2, 3)
In [5]: a.size
Out[5]: 6
```



# NumPy: Array Manipulation

- 2D array: creation
  - Common 2D arrays
    - `np.zeros((N, M), dtype=None)`
    - `np.ones((N, M), dtype=None)`
    - `np.full((N, M), value, dtype=None)`
    - `np.eye(N, M=None, dtype=None)`
  - Diagonal arrays
    - `np.diag(v, k=0)`
    - `v` is 2D array: returns `k`-th diagonal of `v` in 1D array
    - `v` is 1D array: returns 2D array with `v` on `k`-th diagonal

# NumPy: Array Manipulation



- 2D array: indexing
  - Componentwise slicing

```
In [1]: a = np.diag(np.arange(5))
In [2]: a[1]
Out[2]: array([0, 1, 0, 0, 0])
In [3]: a[1, 1]
Out[3]: 1
In [4]: a[1, 2] = 9
In [5]: a[:, 2]
Out[5]: array([0, 9, 2, 0, 0])
In [6]: a[2::3, ::2]
Out[6]: array([[0, 2, 0]])
```

# NumPy: Array Manipulation

Università di Roma Tor Vergata

- 2D array: indexing
  - Fancy indexing: boolean masks or integer lists

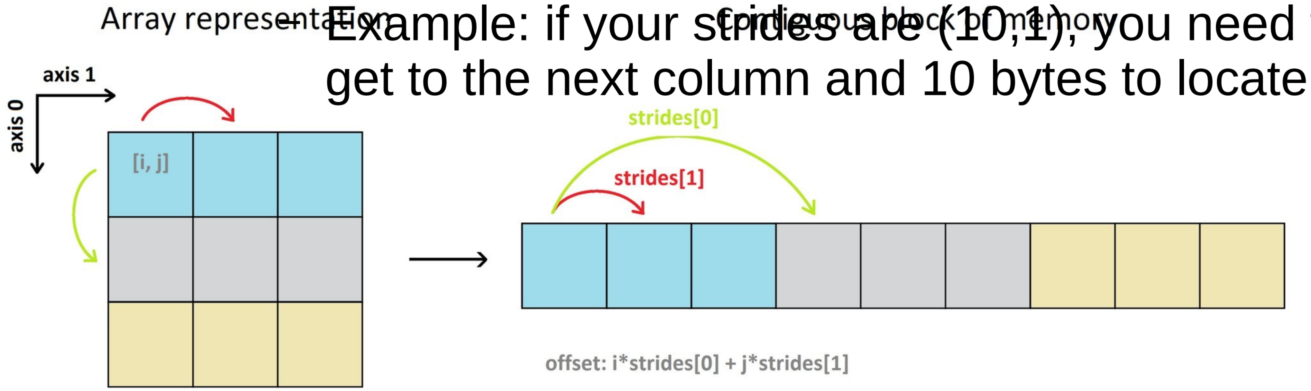
```
In [1]: a = np.diag(np.arange(5))
In [2]: a[a > 0]
Out[2]: array([1, 2, 3, 4])
In [3]: a[[2, 3, 4], [2, 1, 4]]
Out[3]: array([2, 0, 4])
In [4]: a[[2, 3]]
Out[4]: array([[0, 0, 2, 0, 0],
   ...:        [0, 0, 0, 3, 0]])
In [5]: a[[0, 2, 4], 2]
Out[5]: array([0, 2, 0])
```

# NumPy: Array Manipulation



- 2D array: memory
  - Memory structure
    - `data`: pointer indicating the memory address of the first byte in the array
    - `dtype`: pointer describing the data type of objects contained in the array
    - `shape`: tuple indicating the shape of the array
    - `strides`: tuple indicating how many bytes should be skipped in memory to go to the next object in each direction

Example: if your strides are (10,1), you need to proceed one byte to get to the next column and 10 bytes to locate the next row.



# NumPy: Array Manipulation

- 2D array: memory
  - Slicing can be represented by changing shape, strides, and data pointer

```
In [1]: a = np.zeros((10, 20), dtype='int64')
In [2]: a.shape, a.strides
Out[2]: ((10, 20), (160, 8))
In [3]: b = a[::2, ::3]
In [4]: b.shape, b.strides
Out[4]: ((5, 7), (320, 24))
```

- Higher-dimensional array: idem

# NumPy: Array Manipulation



- Shape manipulation
  - Change shape
    - Flattening: `np.ravel(a, order='C')`
    - Reshaping: `np.reshape(a, shape, order='C')`
    - Add a dimension: indexing with `np.newaxis`
    - Similar operators can be applied directly to array
      - example: `a.ravel(order='C')`
  - Change size
    - Use copies when enlarging: `np.resize(a, shape)`
    - Use zeros when enlarging: `a.resize(shape)`
    - Be careful with views!

objects ordered per
row: C-style
col: Fortran-style (F)

# NumPy: Array Manipulation

Università di Roma Tor Vergata

- Shape manipulation
  - View, in-place or copy depends on operation

```
In [1]: a = np.array([[0, 1], [2, 3], [4, 5]])
In [2]: a.ravel()                             # view
Out[2]: array([0, 1, 2, 3, 4, 5])
In [3]: a.reshape((2, -1))      # a.reshape((2, 3))
Out[3]: array([[0, 1, 2],                     # view
   ...:        [3, 4, 5]])
In [4]: a.resize((2,2))                   # in-place
In [5]: a
Out[5]: array([[0, 1],
   ...:        [2, 3]])
```

# NumPy: Array Manipulation



- Shape manipulation
  - Combination of arrays
    - Existing dimension: `np.concatenate((a1, a2), axis=0)`
    - New dimension: `np.stack((a1, a2), axis=0)`
    - Insertion: `np.insert(a, inds, vals, axis=None)`
  - Shrinkage
    - Splitting: `np.split(a, inds, axis=0)`
    - Deleting: `np.delete(a, inds, axis=None)`
  - Repetition
    - Tiling: `np.tile(a, reps)`
    - Repeating: `np.repeat(a, reps, axis=None)`

flattened array:
axis=None
specific dimension:
axis=dim

# NumPy: Array Manipulation



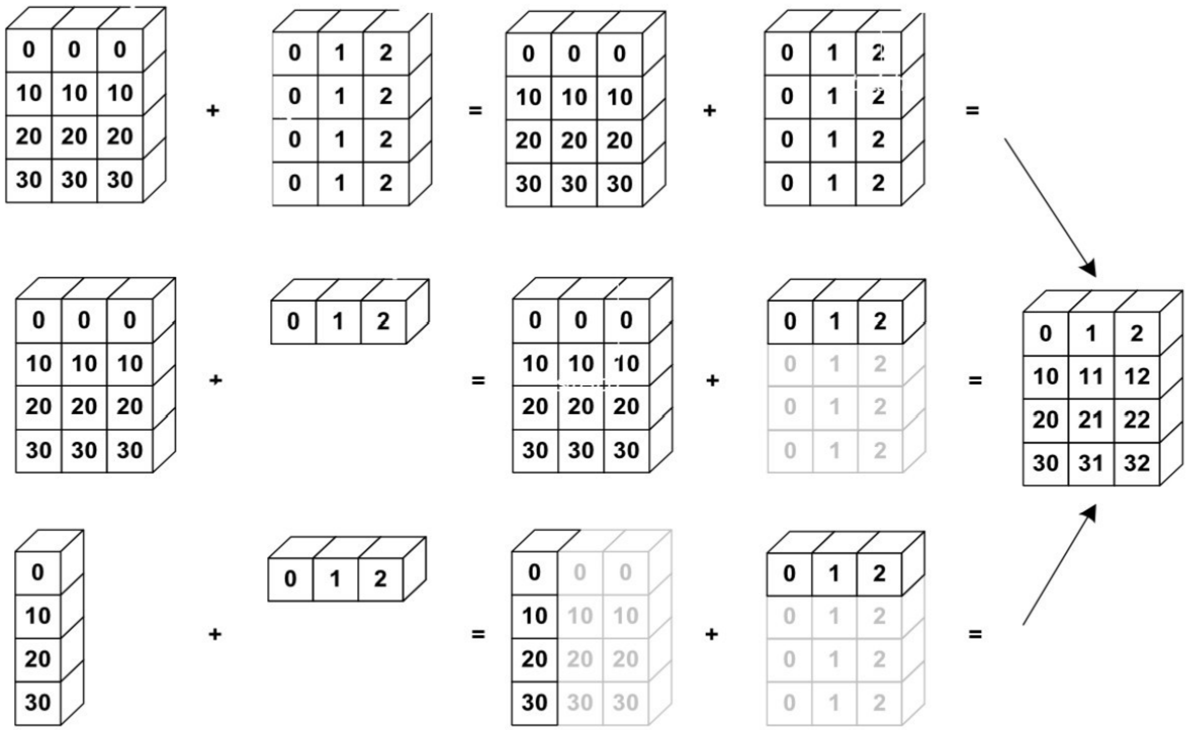
- Mathematical operations
  - Basic arithmetic operations are elementwise
    - Addition (`+`), subtraction (`-`), multiplication (`*`), division (`/`)
    - Power (`**`), integer division (`//`), modulo (`%`)
    - Arrays of same size or scalars

```
In [1]: a = np.array([1, 2, 3, 4])
In [2]: b = np.ones(4) + 1
In [3]: a * b
Out[3]: array([2., 4., 6., 8.])
In [4]: 2**(a + 1) - a
Out[4]: array([3, 6, 13, 28])
```

# NumPy: Array Manipulation

Università di Roma Tor Vergata

- Mathematical operations
  - Broadcasting: arrays are extended so they all have same dimension



# NumPy: Array Manipulation

- Mathematical operations
  - Comparison operations
    - Elementwise: `==`, `!=`, `<`, `<=`, `>`, `>=`
    - Arraywise: `np.array_equal(a, b)`, `np.array_equiv(a, b)`
  - Logical operations
    - `np.logical_and(a, b)`, `np.logical_or(a, b)`, `np.logical_not(a)`
    - Bitwise: `a & b`, `a | b`, `~a`
  - Mathematical functions
    - Power functions: `np.exp(a)`, `np.log(a)`, `np.sqrt(a)`
    - Trig functions: `np.cos(a)`, `np.sin(a)`, `np.tan(a)`
    - Rounding functions: `np.ceil(a)`, `np.floor(a)`, `np.abs(a)`

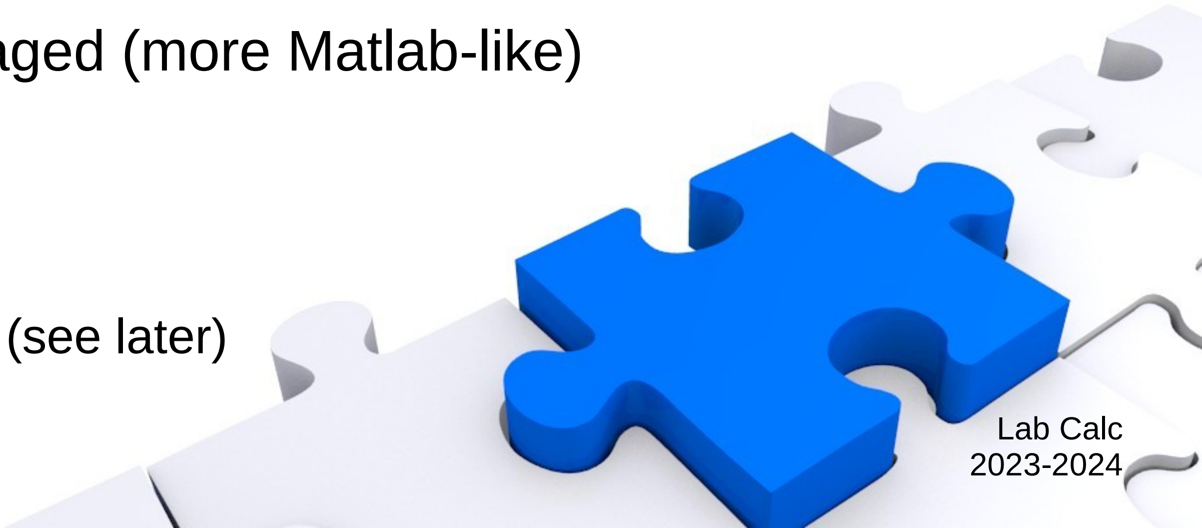
# NumPy: Array Manipulation



- Mathematical operations
  - Reduction operations
    - Sum/product: `np.sum(a, axis=None)`, `np.prod(a, axis=None)`
    - Min: `np.min(a, axis=None)`, `np.argmin(a, axis=None)`
    - Max: `np.max(a, axis=None)`, `np.argmax(a, axis=None)`
    - Logics: `np.all(a, axis=None)`, `np.any(a, axis=None)`
    - Statistics: `np.mean(a, axis=None)`, `np.std(a, axis=None)`
    - Similar operators can be applied directly to array
      - example: `a.sum(axis=None)`
  - Sorting
    - `np.sort(a, axis=-1)`, `np.argsort(a, axis=-1)`

# NumPy: Array Manipulation

- Mathematical operations
  - Linear algebra operations
    - Multiplication: `np.dot(a, b)` or `a.dot(b)` or `a @ b` (since Python 3.5)
    - Transposition: `np.transpose(a)` or `a.T`
    - Trace: `np.trace(a, offset=0)` or `a.trace(offset=0)`
    - Triangle matrices: `np.triu(a, k=0)`, `np.tril(a, k=0)`
  - Note: the class `numpy.matrix` is discouraged (more Matlab-like)
  - Advanced linear algebra packages
    - Basic linear algebra: `numpy.linalg`
    - More efficient linear algebra: `scipy.linalg` (see later)

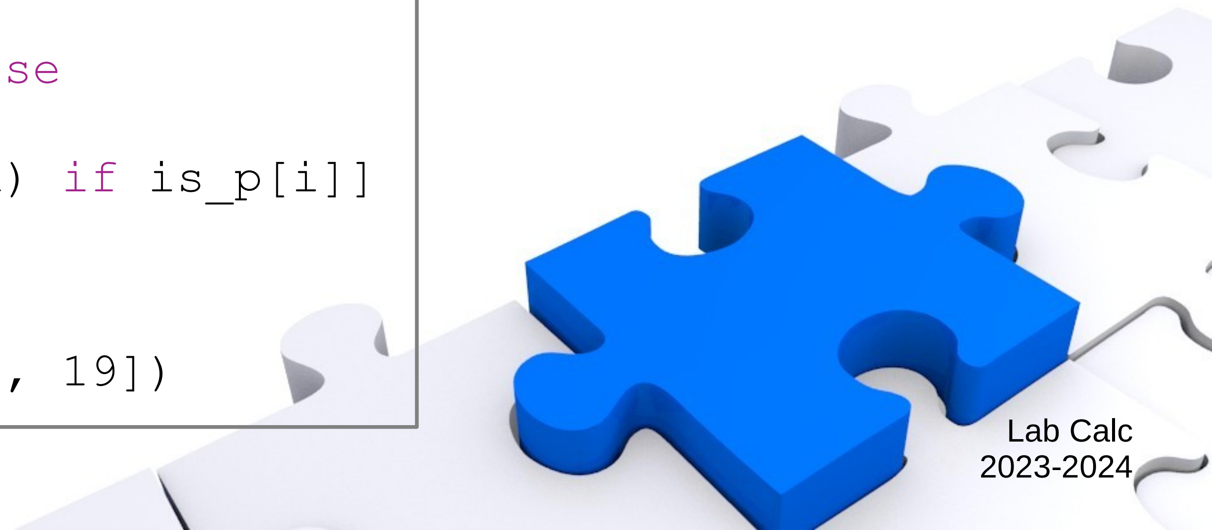


# NumPy: Array Manipulation

- NumPy example
  - Generating all prime numbers (using list)

```
In [1]: def prime_slow_list(n):
   ...:     is_p = [True for i in range(n)]
   ...:     is_p[0] = is_p[1] = False
   ...:     for i in range(2, n):
   ...:         for j in range(2, i):
   ...:             if (i % j == 0):
   ...:                 is_p[i] = False
   ...:                 break
   ...:     l_p = [i for i in range(n) if is_p[i]]
   ...:     return l_p
In [2]: prime_slow_list(20)
Out[2]: array([2, 3, 5, 7, 11, 13, 17, 19])
```

%timeit:
n = 100000
~20 sec per loop



# NumPy: Array Manipulation

Università di Roma Tor Vergata

- NumPy example
  - Sieve of Eratosthenes for prime numbers (using list)

```
In [1]: def prime_sieve_list(n):
   ...:     is_p = [True for i in range(n)]
   ...:     is_p[0] = is_p[1] = False
   ...:     N_max = int(math.sqrt(n - 1)) + 1
   ...:     for i in range(2, N_max):
   ...:         if is_p[i]:
   ...:             for j in range(i*i, n, i):
   ...:                 is_p[j] = False
   ...:     l_p = [i for i in range(n) if is_p[i]]
   ...:     return l_p
In [2]: prime_sieve_list(20)
Out[2]: array([2, 3, 5, 7, 11, 13, 17, 19])
```

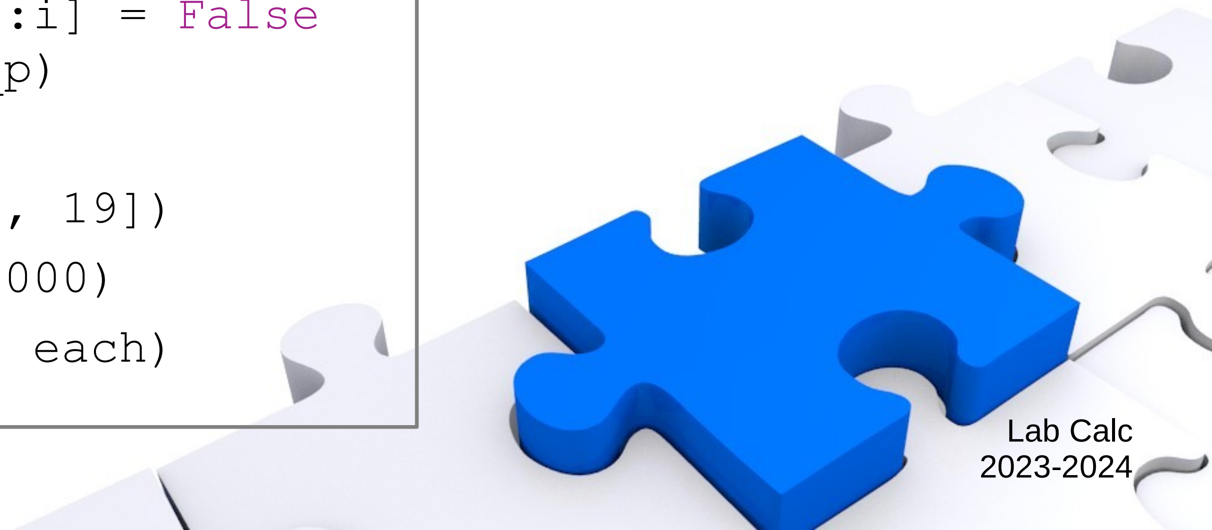
%timeit:
n = 100000
~10 ms per loop

# NumPy: Array Manipulation

- NumPy example
  - Sieve of Eratosthenes for prime numbers (using array)

```
In [1]: def prime_sieve_array(n):
   ...:     is_p = np.ones(n, dtype='bool')
   ...:     is_p[:2] = False
   ...:     N_max = int(np.sqrt(n - 1)) + 1
   ...:     for i in range(2, N_max):
   ...:         if is_p[i]: is_p[i*i::i] = False
   ...:     return np.flatnonzero(is_p)
In [2]: prime_sieve_array(20)
Out[2]: array([2, 3, 5, 7, 11, 13, 17, 19])
In [3]: %timeit prime_sieve_array(100000)
258 µs ± 21.2 µs per loop (1000 loops each)
```

%timeit:
n = 100000
~260 µs per loop



# NumPy: Array Manipulation



- Module `polynomial`
  - Different polynomial representations
    - Power (`Polynomial`), Chebyshev (`Chebyshev`), Legendre (`Legendre`), ...
    - Coefficients represented by list

polynomial
$3x^2+2x-1$

```
In [1]: coef = [-1, 2, 3]
   ...: p = np.polynomial.Polynomial(coef)
In [2]: p.degree()
Out[2]: 2
In [3]: p.roots()
Out[3]: array([-1., 0.33333333])
```

  - Note: the class `numpy.poly1d` is discouraged

# NumPy: Array Manipulation

Università di Roma Tor Vergata

- Module `polynomial`
  - Polynomial operations
    - Evaluation and substitution
    - Standard operations: `+`, `-`, `*`, `**`, `//`, `%`, `==`, `!=`

```
In [4]: p(0)
Out[4]: -1.0
In [5]: q = p(p) + p ** 2
In [6]: q.degree()
Out[6]: 4
In [7]: q(np.arange(3))
Out[7]: array([1., 71., 929.])
```

# NumPy: Array Manipulation

- Module `polynomial`
  - Polynomial operations
    - Indefinite integral (`integ`) and derivative (`deriv`)
    - Polynomital fitting (`fit`)

```
In [1]: x = np.linspace(0, 1, 20); y = np.sin(x)
In [2]: f = np.polynomial.Polynomial.fit(x, y, 3)
In [3]: f(0)
Out[3]: -0.00018474606249202496
In [4]: g = f.integ().deriv()
In [5]: g == f
Out[5]: True
```

# NumPy: Array Manipulation

- Input and output
  - Text files

```
In [1]: data = np.ones((3, 3))
In [2]: np.savetxt('datafile.txt', data)
In [3]: data3 = np.loadtxt('datafile.txt')
```

  - Binary files

```
In [1]: data = np.ones((3, 3))
In [2]: np.save('datafile.npy', data)
In [3]: data2 = np.load('datafile.npy')
```