NumPy Notes

February 15, 2022

1 Introduction to NumPy

NumPy (**Num**erical **Py**thon) is the fundamental package used for scientific computing in Python. Numpy offers a number of key features for scientific computing, in particular *multi-dimensional* arrays (or **ndarrays** in NumPy speak) such as vectors or matrices, as well as the attendant operations on these objects.

Three main reasons why NumPy is so useful for scientific computing and appears so often in data science are:

- Speed: for example, using NumPy arrays can be *ten times* faster than Python's lists. This occurs because NumPy's arrays are *fixed in size*, whereas lists can *change in size*. As we shall see, the elements in NumPy arrays must all be the *same type* (ints or floats, for example), unlike lists where elements can be *different kinds*.
- Functionality: capable of performing a huge number of fast operations on arrays, some of which we will encounter here.
- Many packages in Python rely on NumPy. In fact, Pandas are built on top of NumPy.

In this part of the course, we will only scratch the surface of NumPy's functionality, but as with all things in computer programming, the more you use NumPy the more you will learn! For (much) more information, see the online NumPy Documentation.

1.1 Importing NumPy

Before creating arrays, we need to import the NumPy package. Typically, we import the package as *np*, then use np to access functions from NumPy.

[2]:
```
import numpy as np
```

1.2 Vectors

[3]:
```
# Create vector and query the resulting objects type and dimensions

A = np.array([17, 1, 156]) #Note: a common error is to write: A = np.array(17,␣
 ↪1, 156). What happens when you do this?
print(type(A)) #Note: this creates a "column" vector.
print(A.shape) # Tells us the ndarray is of dimension 3 (x1)
```

```
<class 'numpy.ndarray'>
(3,)
```

Question: what do the operations A.ndim and A.size do?

In a similar way to lists, we can access the elements of the array using brackets. Notice that, unlike *MATLAB*, array indices start from 0 (rather than 1).

[4]:
```
print(A[0], A[1], A[2])
```

```
17 1 156
```

ndarrays are so-called *mutable* objects, meaning we can change their elements.

[5]:
```
A[0] = 49
print(A)
```

```
[ 49   1 156]
```

1.3 Matrices

We can create matrices in much the same way as vectors. Again, be careful with the syntax here: array converts sequences (signified by the use of square brackets) of sequences into two-dimensional arrays.

[6]:
```
B = np.array([[3, 18, 4], [21, 1, 46]])
print(B)
```

```
[[ 3 18  4]
 [21  1 46]]
```

Question: how would you construct a three-dimensional array?

To access entries of the matrix, we again use the square-bracket syntax, specifying the row *then* column. Remember that NumPy indexing starts from zero!

[7]:
```
B[0, 2]
```

[7]:
```
4
```

[8]:
```
B[0, 2] == 4
```

[8]:
```
True
```

We can perform operations on ndarrays, such as taking the tranpose.

```
[9]: # Take the tranpose of B

np.transpose(B)
```

```
[9]: array([[ 3, 21],
            [18,  1],
            [ 4, 46]])
```

1.4 A note on data types

Unlike lists, which can take in multiple data types, ndarray requires all the entries to be of the same data type. In general, when using array to define a new NumPy array, you should pay attention to the data type of the elements in the array. If you don't, and perform calculations with mismatching data types, you might end up with unwanted results! For more on this, see here.

```
[21]: lst_1 = [27.3, "cat", [14, 5, 2]]
print(lst_1)
print(type(lst_1))
```

```
[27.3, 'cat', [14, 5, 2]]
<class 'list'>
```

```
[29]: dt_1 = np.array([14, 5, 2]) # Python assigns the data type, in this case a␣
 ↪64-bit integer...
print(dt_1.dtype)

dt_2 = np.array([14.0, 5.0, 2.0]) #...and in this case, a 64-bit floating point␣
 ↪number
print(dt_2.dtype)

#Alternatively, we can tell Python what type of object is in array...
dt_3 = np.array([11, 21], dtype=np.int64)
print(dt_3.dtype)

# but we can also force it to be of a certain type.
dt_4 = np.array([11.1, 12.7], dtype=np.int64)
print(dt_4.dtype)
print()
print(dt_4)
```

```
int64
float64
int64
int64

[11 12]
```

Note: If you an array that included int64 and float64 numbers - the int64 numbers would be "upcast" to float64 numbers to preserve accuracy.

1.5 In-built arrays

NumPy provides functions that create many commonly used arrays in scientific computing, rather than laboriously typing out all the elements.

[39]:
```
# Create a 4x3 array (4 rows, three columns) containing all zeros:

zeros = np.zeros((4,3)) # Note that the input is a tuple here...why?
print(zeros)
```

```
[[0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]]
```

[40]:
```
# Similarly, create an array containing the same value. (Alternatively fulls =␣
↪np.full((4,3), 0)

fulls = np.full((4,3), 1) #Alternatively, np.ones((4,3))
print(full)
```

```
[[1 1 1]
 [1 1 1]
 [1 1 1]
 [1 1 1]]
```

[42]:
```
# Create matrix with ones on the main diagongal.

identity = np.eye(3,4) # Note that if the syntax np.eye(N) is used, this will␣
↪produce a SQUARE identity matrix of dimension NxN
print(identity)
```

```
[[1. 0. 0. 0.]
 [0. 1. 0. 0.]
 [0. 0. 1. 0.]]
```

[50]:
```
# Create a matrix whose entries are drawn from the half interval [0.0,1.0)

random = np.random.rand(4,3)
print(random)
```

```
[[0.03330037 0.65518479 0.00257132]
 [0.14956798 0.63897228 0.41769437]
 [0.12975192 0.63727318 0.48141184]
 [0.37244747 0.20033267 0.83771455]]
```

1.6 ndarray slice indexing

In a similar way to strings and lists, we can use *slice indexing* to pull out subsets of those data structures. (Essentially just means extracting part of the array.)

```
[51]: string = "Bristol"
      string[1:4]
```

```
[51]: 'ris'
```

Important: The slicing syntax is a little funky. In the case of a vector, the command a[m:n] will pull out the entries m to *n-1* (not n). If you want to pull out all the values starting from index 0 to n - 1, you just have to type a[:n]. If you make n any number greater than the size of the vector, it will pull out all the values from the starting index.

```
[64]: # Create an array of random integers

      randInt = np.random.randint(10, size = (4,3)) # The syntax requires we specify␣
       ↪the highest integer we want, and the size of the matrix
      print(randInt)
```

```
[[7 8 0]
 [5 8 3]
 [8 1 6]
 [0 4 1]]
```

```
[65]: # Pull out first two rows.
      randInt[:2]
```

```
[65]: array([[7, 8, 0],
             [5, 8, 3]])
```

```
[66]: # Pull out last two columns.

      randInt[:,1:3]
```

```
[66]: array([[8, 0],
             [8, 3],
             [1, 6],
             [4, 1]])
```

```
[68]: # How to extract 2x2 square matrix?

      randInt[2:5, 1:3]
```

```
[68]: array([[1, 6],
             [4, 1]])
```

Important note on memory allocation: If you want to extract elements from an array and *create a new array*, you have to be a bit careful. Consider the following example based on the matrix randInt above:

```
[69]: randInt_slice = randInt[:,1:3]
      print(randInt_slice)
```

```
[[8 0]
 [8 3]
 [1 6]
 [4 1]]
```

```
[70]: randInt_slice[0,0] = 25
```

```
[71]: randInt
```

```
[71]: array([[ 7, 25,  0],
             [ 5,  8,  3],
             [ 8,  1,  6],
             [ 0,  4,  1]])
```

Note that changing a value in the extracted array, randInt_slice, *has changed the corresponding value* in the randInt. This happens because randInt_slice points to the *same elements in memory* as randInt. Be especially careful of memory allocation when passing vectors/matrices to functions. Further, randInt_slice has its own indices that are different to randInt. If you wanted the slice to be a complete (deep) copy, you have to make a whole other array as follows:

```
[77]: randInt_slice = np.array(randInt[:,1:3])
      print(randInt_slice)
      print()

      randInt_slice[0,1] = 99
      print(randInt_slice)
      print()
      print(randInt)
```

```
[[25  0]
 [ 8  3]
 [ 1  6]
 [ 4  1]]

[[25 99]
 [ 8  3]
 [ 1  6]
 [ 4  1]]

[[ 7 25  0]
 [ 5  8  3]
 [ 8  1  6]
 [ 0  4  1]]
```

1.7 Basic array operations

NumPy has an enormous amount of everyday operations to perform on arrays. Here are just a few.

[84]:
```
x = np.array([[53, 74], [2, 14]], dtype = np.int)
y = np.array([[31.3, 0.8],[12.7, 8.1]], dtype = np.float64)

print(x)
print()
print(y)
```

```
[[53 74]
 [ 2 14]]

[[31.3  0.8]
 [12.7  8.1]]
```

[85]:
```
print(x + y) # Note that the int got upcast (a method that converts to the more␣
 ↪precise or general definition of number) to floating point to preserve␣
 ↪accuracy
```

```
[[84.3 74.8]
 [14.7 22.1]]
```

[86]:
```
print(x - y)
```

```
[[ 21.7  73.2]
 [-10.7   5.9]]
```

[87]:
```
print(x * y) # Note: for vectors, this is NOT the dot/scalar product (clearly␣
 ↪as it didn't produce a scalar). Rather it is the element-by-element␣
 ↪multiplcation of the elements of x and y
```

```
[[1658.9   59.2]
 [  25.4  113.4]]
```

[90]:
```
print(np.sqrt(x))
```

```
[[7.28010989 8.60232527]
 [1.41421356 3.74165739]]
```

[91]:
```
print(np.sum(x, axis = 0)) #sum over rows
```

```
[55 88]
```

[92]:
```
print(np.sum(x, axis = 1)) #sum over cols
```

```
[127  16]
```

[93]:
```
print(x @ y) #matrix product
```

```
[[2598.7  641.8]
 [ 240.4  115. ]]
```

1.8 Broadcasting

On occasion, you might want to perform operations on matrices with different sizes. However, the dimensions of your matrices should be compatible. The way NumPy handles this is through the process of broadcasting. The easiest example of broadcasting is when you multiply an array by a scalar as follows:

[13]:
```
5 * np.ones([5,5])
```

[13]:
```
array([[5., 5., 5., 5., 5.],
       [5., 5., 5., 5., 5.],
       [5., 5., 5., 5., 5.],
       [5., 5., 5., 5., 5.],
       [5., 5., 5., 5., 5.]])
```

As you can see, each element in the array has been multiplied by the scalar (5). Here is a more complex example.

[14]:
```
# Example

A = np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]])
B = np.array([0,1,0,2])

print(A + B)
```

```
[[ 1  3  3  6]
 [ 5  7  7 10]
 [ 9 11 11 14]]
```

In the example above, we have a 3x4 matrix and 4x1 vector. Clearly the operation A + B makes no strict mathematical sense. However, *broadcasting assumes the operation you are trying to perform*, in this case adding the vector B to each of the rows of the matrix A. Here NumPy recognizes that the number of rows of B matches the number of columns of A, so adds B to each row of A.

In general, the NumPy documentation advises us:

"When operating on two arrays, NumPy compares their shapes *element-wise*. It starts with the trailing dimensions and works its way forward. Two dimensions are *compatible* when

- they are equal, or
- one of them is 1

Here's an example that doesn't work. Why not?

[15]:
```
C = np.array([4, 5, 1])
print(A + C)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-15-8d8cd82f0570> in <module>
      1 C = np.array([4, 5, 1])
----> 2 print(A + C)

ValueError: operands could not be broadcast together with shapes (3,4) (3,)
```

1.9 End

I hope this has given you some idea of the breadth of functionality available with NumPy. In the Worksheet, you will put much of this and more into practice, while learning how to search and read NumPy documentation.

2 Introduction to SciPy

SciPy is a catalogue of algorithms and functions that are built on NumPy.

2.1 Solving differential equations with SciPy

[17]:
```
import numpy as np

import matplotlib as mpl

import matplotlib.pyplot as plt

from scipy import integrate, linalg, optimize #integration and odes, linear␣
↪algebra, optimization/root-finding

from scipy.integrate import solve_ivp # import solve_ivp from scipy integrate

from scipy.linalg import solve # solve function for later...
```

Using the function **solve_ivp**, SciPy can solve differential equations of the form

$\frac{\mathrm{d}\mathbf{y}}{\mathrm{d}t} = \mathbf{f}(\mathbf{y}, t)$ subject to $\mathbf{y}(0) = \mathbf{y}_0$,

where $\mathbf{y}$, $\mathbf{f}$, and $\mathbf{y}_0$ are vectors.

Before we do so, we need to import solve_ivp.

[18]:
```
from scipy.integrate import solve_ivp
```

Look at the SciPy documentation for solve_ivp to see what inputs you need to provide.

Solving an SIR model

The SIR model is a rudimentary differential equation model for the number of people who are susceptible (S), ill (I), and have recovered (R) from the spread of a contagion over time. One version of an SIR is as follows:

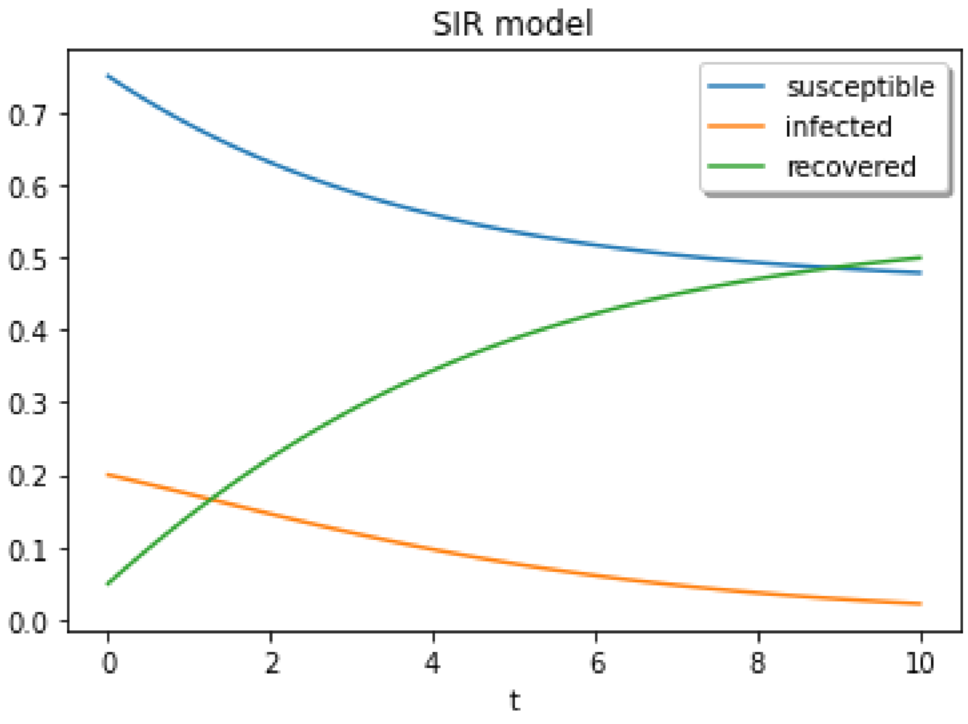
$$\frac{\mathrm{d}S}{\mathrm{d}t} = -\beta SI, \tag{1}$$
$$\frac{\mathrm{d}I}{\mathrm{d}t} = \beta SI - \gamma I, \tag{2}$$
$$\frac{\mathrm{d}R}{\mathrm{d}t} = \gamma I. \tag{3}$$
$$\tag{4}$$

In this case, $\mathbf{y} = (S, I, R)^T$ and $\mathbf{f} = (-\beta SI, \beta SI - \gamma I, \gamma I)^T$. Notice that adding these three equations gives $\frac{\mathrm{d}}{\mathrm{d}t}(S + I + R) = 0$.

Let's see how to set up a solver for solving the SIR model.

[22]:
```
# Define parameters and create a function that returns the right hand side of␣
↪the differential equations.

beta = 0.5
gamma = 0.5

def SIRode(t, y):
        return [-beta*y[0]*y[1], beta*y[0]*y[1] - gamma*y[1], gamma*y[1]]

# Next, create a vector of initial conditions (percentage of people who are S,␣
↪I, and R)

y0 = [.75, .20, .05]

# Set the time span for the simulation

tspan = [0, 10]

# Finally, invoke the solve_ivp function to solve the differential equations.

sol = solve_ivp(SIRode, tspan, y0, t_eval = np.linspace(0, 10, num = 101))

# Plot solutions

plt.plot(sol.t, sol.y.T)
plt.legend(['susceptible', 'infected', 'recovered'], shadow=True)
plt.xlabel('t')
plt.title('SIR model')
```

[22]:
```
Text(0.5, 1.0, 'SIR model')
```



Try some other parameters and initial condtions and see what happens! Do the results correspond to your intuition?

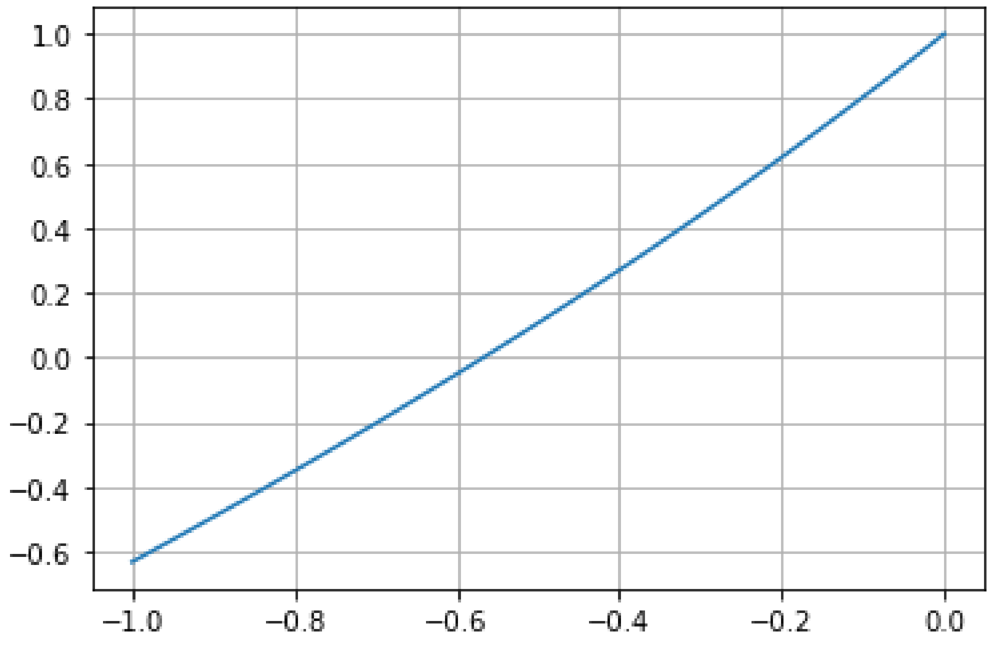
2.2 Root-finding

The optimization toolbox in SciPy provides many off-the-shelf optimization algorithms. We'll demonstrate finding the roots of a nonlinear function $x + \exp(x)$.

[33]:
```
from scipy.optimize import root

def func(x):
    return x + np.exp(x)

sol = root(func, 0)

sol.x
```

```
[33]: array([-0.56714329])
```

```
[36]: x = np.linspace(-1, 0, num = 101)
      y = x + np.exp(x)

      plt.plot(x,y)
      plt.grid()
```



2.3 Linear algebra

SciPy has loads of functionality for linear & nonlinear algebra problems. Note that NumPy also has the former which perform just as well as SciPy for the kind of problems we'll look at, but SciPy has some better features under the hood that we won't go into here.

Example: Find the cubic polynomial that passes through the points $(x, y) = (-3, 6), (-1, 3),\ (0, 0),\ (5, 4)$.

The general form of a cubic polynomial is $y = ax^3 + bx^2 + cx + d$. Inserting the points into this equation gives four equations for four unknowns.

$$-27a + 9b - 3c + d = 6, \tag{5}$$
$$-a + b - c + d = 3, \tag{6}$$
$$d = 0, \tag{7}$$
$$125a + 25b + 5c + d = 4. \tag{8}$$

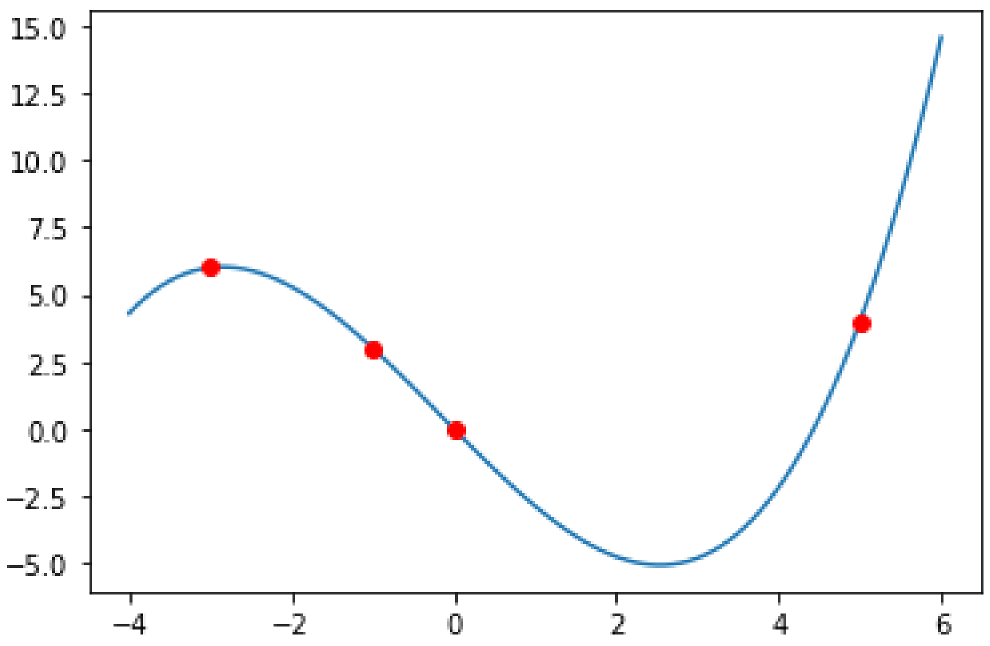
We can re-write this as a matrix vector problem for the unknowns (a, b, c, d).

```
[10]: pts = np.array([[-3,6],[-1,3],[0,0],[5,4]]) # array of (x,y) points

      A = np.array([[-27, 9, -3, 1],[-1, 1, -1, 1],[0, 0, 0, 1],[125, 25, 5, 1]])
      b = np.array([6, 3, 0, 4])

      x = linalg.solve(A,b)
      print(x)
```

```
[ 0.14166667  0.06666667 -3.075       0.        ]
```

```
[11]: xdom = np.linspace(-4,6,num=101,endpoint = True)
      y = x[0]*xdom**3 + x[1]*xdom**2 + x[2]*xdom + x[3]
      plt.plot(xdom,y)
      plt.plot(pts[:,0],pts[:,1],'ro')
```

```
[11]: [<matplotlib.lines.Line2D at 0x7f974c2766a0>]
```



```
[ ]:
```