NumPy/SciPy Tutorial Script

Software Carpentry Workshop, 2014, NOCS, Southampton

Created by Bogdan Vera, based on original SWC Lecture Series

1 WHAT IS NUMPY?

Hello, and welcome to Software Carpentry's lecture series on matrix programming in NumPy. In this episode, we will talk about three ways to work with numerical data, and we will explain why a numerical package like NumPy is the best option.

NumPy is a python library that allows matrix operations and algebra. It allows use to write programs that perform mathematical operations without having to write a large number of loops or other programming constructs that obscure that actual operations.

SciPy is a library that, together with NumPy, allows Python to be used similarly to Matlab; It provides a huge amount of functionality relating to statistics, linear algebra, mathematical optimization and much more. Furthermore, using the *matplotlib* library we can create plots and visualizations of data.

2 BASIC OPERATIONS

First we have to import the library. Start Python in the command line (or IPython if you've got it).

```
>>> import numpy as np
```

Now let's create a vector of data.

```
>>> vals = [1 ,2 ,3]
```

```
>>> arr = np.array(vals)
```

This creates a Numpy Array containing the numbers 1 2 and 3. Unlike python lists, numpy arrays are homogenous, such that all values must have the same type. This saves memory and is faster to process as it reduces python run-time overhead.

If we specify different types, Numpy finds the most general type and uses that.

```
>>> arr = np.array([1, 2.3])
```

Or you can specify the type:

```
>>> arr = np.array([1, 2, 3 ,4], dtype=float32)
```

In NumPy there are a number of basic data types:

```
Bool, int, int8, int16, int32, int64, uint[8-64], float[32,64,128],
complex, complex[64,126]
```

A number of functions are available to create arrays:

```
>>> z = np.zeros([2,3])
>>> z
array([[0., 0., 0.], [0., 0., 0.]])
```

This has created a 2x3 matrix with zeroes for all elements. Note how the matrix is displayed as two row vectors separated by comma.

Similarly we can generate a matrix of all ones using np.ones() and an identity matrix using np.identity().

Assigning does not copy data, it aliases the data (passing by reference).

```
>>> first = numpy.ones([2,2])
>>> first
array([[1., 1.], [1., 1.]])
>>> second = first
>>> second[0, 0] = 9
>>> first
array([[9., 1.], [1., 1.]])
```

We CAN copy a matrix:

```
>>> second = first.copy()
```

Numpy arrays have properties:

```
>>>first.shape
(2,2)
>>> block = numpy.zeros([4,7,3])
>>> block.shape
(4,7,3)
```

Note the lack of parantheses; shape is a parameter/variable, not a method call.

More functions:

```
>>> first.transpose()
>>> first.size() %total number of elements
```

```
>>> first.ravel() %flattens a matrix by concatenating rows
```

We can reshape explicitly:

```
>>> first = numpy.array([1,2,3,4,5,6])
>>> first.shape
(6,) -> tuple with one element
>>> second = first.reshape(2,3)
>>> second
array([[1,2,3],[4,5,6]])
```

Number of elements must be the same!

3 MATRIX OPERATIONS

We can add matrices together, or perform a number of operations. Unlike python lists, adding or multiplying numpy arrays/matrices results in an expected mathematical results (not concatenation or repetition of lists).

```
>>> A = np.array([[1,2,3],[1,4,2]])
>>> B = np.array([[1,2,1],[1,1,2]])
>>> A+B %adding matrices
>>> A*B %element-wise multiplication
>>> C = np.dot(A,B) %matrix multiplication (dot product)
```

Indexing is simple and similar to matlab.

```
>>> vector = array([10,20,30,40])
>>> vector[0:3] = 1
vector
array([1,1,1,40])
```

We can index by subscript arrays:

```
>>> vector = array([10,20,30,40])
>>> subscript = [3,1,2]
>>> vector [subscript]
array([40,20,30])
```

Needless to say, Python indexes from 0, unlike Matlab! You'll have to change your indexing if you port from Matlab.

We can do this in multiple dimensions:

```
>>> square = np.array([[5,6],[7,8]])
>>> square[[1]]
array([[7,8]])-> second row returned
>>> A = np.array([[1,2,3],[2,6,4]])
>>> A([1,0:2]) %second row, columns 1 to 3
array([2,6])
```

We can do Boolean operations between arrays.

```
>>> vector = np.array([0,10,20,30])
>>>vector<25
array([ True, True, False, False], dtype=bool)
```

We can use Boolean subscripts:

```
>>> vector[vector<25]
array([0,10,20]) -> all elements smaller than 25
```

More:

```
>>> (vector <= 20) & (vector>=20)
array([False, False, True, False], dtype=bool)
```

4 MATHS AND LINEAR ALGEBRA

Numpy provides a large number of out of the box functions. They all work in intuitive ways:

```
>>> A = np.array([[1,2,3],[2,6,4]])
>>> np.mean(A,1) -> mean along the first dimension
>>> np.sum(A) -> sum
>>> np.diff(A) -> derivative/finite difference
>>> np.diff(A,2) -> second order derivative
>>> np.diff(A,2) -> second order derivative
>>> np.diff(A,axis=1, n=1) -> second order derivative along second
axis
```

Use documentation to look at other functions: fft, convolve, diagonal, roots, solve, svd, etc.

5 PLOTTING

We will use another Python package to do scientific plotting.

```
>>> from  matplotlib import pyplot as plt
```

We imported matplotlib's pyplot module, aliased as plt. We can now do various types of data plotting. We can use the same data reading methodology as in the Python session, using the reader function and iterating through a file line by line, parsing the data into the program.

Some forms of data have a standardized structure, and there are prebuilt functions to load this data directly into numpy. For example, there is an established method to load a csv file into a numpy array.

```
>>>from numpy import genfromtxt
```

Genfromtxt can directly parse a text file of any type based on a delimiter. Let's parse our data file, 'timeseries.csv' into an array:

```
>>>ts = genfromtxt('timeseries.csv', delimiter=',')
```

Columns are split by comma. We could have used another delimiter. If our files have a complex delimiting style, we can write our own function to parse line by line. Most data will come in easily parsable form.

We will see how to parse heterogenous data in the Panda tutorial by Devasena.

Let's plot our time series.

Ask students: what is 'ts'? It's a numpy array. First column is time, second column is a measurement. We want to plot the measurements over time. We can plot in 2d one set of values over another by saying:

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

Our data is combined into one array so we need to plot one column over the other:

```
>>>plt.plot(ts[0,:], ts[1,:])
```

We have a nice plot. We can zoom and pan around.

```
>>>plt.title('Time Series Plot')
```

```
>>>plt.xlabel('Time')
```

```
>>>plt.ylabel('Density')
```

```
>>>plt.ylim([0 50])
```

```
>>>plt.xlim([20 100])
```

Let's try some other data visualizations:

```
>>>plt.boxplot(ts[1,:])
```

```
>>>plt.hist(ts[1,:])
```

Exercise:

Challenge: use Numpy to smooth the time series with a running average: average together the last up to 5 samples and the current one to calculate the current sample.

F[k] = (f[k-5] + f[k-4] + f[k-3] + f[k-2] + f[k-1] + f[k])/6.0

find the maximum value of the time series and its location (argmax). Then take a window of 100 samples around this point and plot the original data around this point.

You can either write the code yourself or use Numpy functions.

6 Working with some real data (Passing on to Devasena for Big Data Stuff)