Numpy Crash Course

"NumPy is the fundamental package for scientific computing in Python. It is a Python library that provides a multidimensional array object, various derived objects (such as masked arrays and matrices), and an assortment of routines for fast operations on arrays, including mathematical, logical, shape manipulation, sorting, selecting, I/O, discrete Fourier transforms, basic linear algebra, basic statistical operations, random simulation and much more"

You can visit the full documentation here (https://docs.scipy.org/doc/numpy-1.10.1/user/whatisnumpy.html)

Importing Numpy

The `np` is a very popular alias given to numpy

In [5]:
```
import numpy as np
```

Let's run through an example showing how powerful NumPy is.
Suppose we have two lists `a` and `b`, consisting of the first `100,000 non-negative numbers`, and we want to create a new list c whose ith element is `a[i] * b[i]`.

Approach Without NumPy:

Using Python lists

In [6]:
```
%%time
a = [i for i in range(100000)]
b = [i for i in range(100000)]
```

```
Wall time: 21 ms
```

In [7]:
```
%%time
c = []
for i in range(len(a)):
    c.append(a[i]  *  b[i])
```

```
Wall time: 26 ms
```

That's the thing we want you to notice the real time difference. The Wall Time which a process needs to complete its task .

- 1st : Wall time: 12 ms
- 2nd : Wall time: 36 ms.

***Note:** The `%%time` is the magic command for calculating the execution time of the cell.*

Using Numpy

In [8]:
```
%%time
a = np.arange(100000)
b = np.arange(100000)
```

```
Wall time: 4 ms
```

In [9]:
```
%%time
c = a  * b
```

```
Wall time: 2 ms
```

The result is 10 to 15 times faster, and we could do it in fewer lines of code (and the code itself is more intuitive)

Regular Python is much slower due to type checking and other overhead of needing to interpret code and support Python's abstractions.

For example, if we are doing some addition in a loop, constantly type checking in a loop will lead to many more instructions than just performing a regular addition operation. NumPy, using optimized pre-compiled C code, is able to avoid a lot of the overhead introduced.

The process we used above is vectorization. Vectorization refers to applying operations to arrays instead of just individual elements (i.e. no loops).

Why vectorize?

1. Much faster
2. Easier to read and fewer lines of code
3. More closely assembles mathematical notation

Vectorization is one of the main reasons why NumPy is so powerful.

What is an Array

A numpy array is a grid of values, all of the same type, and is indexed by a tuple of non-negative integers. The number of dimensions is the rank of the array; the shape of an array is a tuple of integers giving the size of the array along each dimension.

Creating A simple array of 3 integers

In [9]:
```
array = np.array([1,2,3])
print(array)
```

```
[1 2 3]
```

`Item Size` **Itemsize is the size of one element,this gives 4 cause u have integers and so**

each item = 4 bytes.

In [10]:
```
array.itemsize
```

Out[10]:
```
4
```

You can also get an array of a range using the `arange` funciton

In [11]:
```
array = np.arange(10)  # ...means give me an array 0-9
array
```

Out[11]:
```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Check size of array

In [12]:
```
array.size
```

Out[12]:
```
10
```

Now its better to store things as numpy array cause in array 1 element = 4 bytes but in python 1 object = 14 bytes Numpy array is also fast cause it will take a lot less time to process when you have large data.

In [13]:
```
# To create an array of size 3-5.
np.arange(3,6)
```

Out[13]:
```
array([3, 4, 5])
```

Getting Length of Array

In [14]:
```
len(array)
```

Out[14]:
```
10
```

Getting Shape

In [15]:
```
array.shape
```

Out[15]:
```
(10,)
```

Type

In [16]:
```
print(type(array))
```

```
<class 'numpy.ndarray'>
```

Get Datatype of Array

In [17]:
```
array.dtype
```

Out[17]:
```
dtype('int32')
```

Changing Datatype of Array

In [21]:
```
float_array = np.array([2,3,4,5,6,7], dtype=np.float64)
int64_array = np.array([1, 2], dtype=np.int64)
print(float_array.dtype)
print(int64_array.dtype)
```

```
float64
int64
```

Indexing in Array

Example: print 1st 4th and 6th element

In [22]:
```
print(array[0], array[3], array[5])
```

```
0 3 5
```

Modifying Array elements

In [23]:
```
array[0] = 100
print(array)
```

```
[100   1   2   3   4   5   6   7   8   9]
```

Range with Step Size

In [24]:
```
#Now take a step of 2 on each step.
np.arange(2,11,2)
```

Out[24]:
```
array([ 2,  4,  6,  8, 10])
```

About N-Dimensional Arrays

ndarrays, n-dimensional arrays of homogenous data type, are the fundamental datatype used in NumPy. As these arrays are of the same type and are fixed size at creation, they offer less flexibility than Python lists, but can be substantially more efficient runtime and memory-wise. (Python lists are arrays of pointers to objects, adding a layer of indirection.)

The number of dimensions is the rank of the array; the shape of an array is a tuple of integers giving the size of the array along each dimension.

1 Dimensional or Zero Rank Array

In [25]:
```
a = np.array([3,3,0,3,3]) #1D array
a
```

Out[25]:
```
array([3, 3, 0, 3, 3])
```

Get Dimensions of Array

In [26]:
```
a.ndim
```

Out[26]:
```
1
```

Creating a 2D Array

In [27]:
```
b=np.array([[2,3],[4,5],[6,7]]) # 2D array
print(b.ndim)
print(b.shape) # returns rows,columns
b
```

```
2
(3, 2)
```

Out[27]:
```
array([[2, 3],
       [4, 5],
       [6, 7]])
```

Indexing a 2D array

You first pass in the row number than the col number

In [28]:
```
print(b[0, 0], b[0, 1], b[1, 0])
```

```
2 3 4
```

Alternative

In [29]:
```
print(b[0][0], b[0][1], b[1][0])
```

```
2 3 4
```

Creating 3D arrays

In [30]:
```
b = np.array([[[1],[2],[3]],[[4],[5],[6]]])   # Create a rank 3 array
print (b)
print(b.ndim)
print(b.shape)
```

```
[[[1]
  [2]
  [3]]

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

And so on you can Create N dimensional Arrays