

DATA SCIENCE INFINITY



NumPy

FOR

DATA SCIENCE

AN OVERVIEW

Let's start at the top...



NumPy is a portmanteau from "**Numerical Python**"



NumPy contains a broad array of functionality for **fast** numerical & mathematical operations in Python



The core data-structure within NumPy is an **ndArray** (or n-dimensional array)



Behind the scenes - much of the NumPy functionality is written in the programming language **C**



NumPy functionality is used in other popular Python packages including **Pandas**, **Matplotlib**, & **scikit-learn**!

Don't be fooled...

At first glance, a **NumPy array** resembles a **List** (or in the case of multi-dimensional arrays, a List of Lists...)

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

→ List

```
>> array([1, 2, 3])
```

→ 1 Dimensional Array

```
>> [[1, 2, 3], [4, 5, 6]]
```

→ List of Lists

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

→ 2 Dimensional Array

The similarities however are mainly superficial!

NumPy provides us the ability to do so much more than we could do with Lists, and at a **much, much faster speed!**

How is NumPy so fast?



NumPy is written mostly in **C** which is much faster than Python



NumPy arrays are used with **homogenous** (same) data types only, whereas lists (for example) can contain any data type. This fact means NumPy can be more **efficient** with its memory usage



NumPy has the ability to divide up sub-tasks and run them in parallel!

Getting started...

Installing NumPy

If you have Python, you can install NumPy using the following command...

```
pip install numpy
```

Note - if you're using a distribution such as **Anaconda** then NumPy comes pre-installed!

Importing NumPy

To ensure you can access all of the amazing functionality from within NumPy - you import it like so...

```
import numpy as np
```

Creating Arrays I

Creating a 1-D array

```
np.array([1,2,3])  
>> array([1, 2, 3])
```

Creating a 2-D array

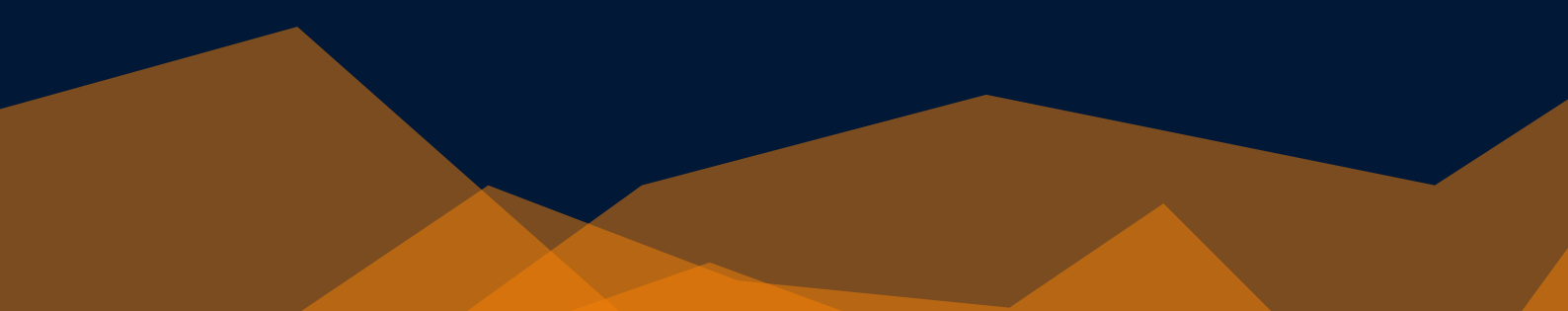
```
np.array([[1,2,3],[4,5,6]])  
>> array([[1, 2, 3],  
          [4, 5, 6]])
```

Creating an array filled with zeros

```
np.zeros(5)  
>> array([0., 0., 0., 0., 0.])
```

Creating an array filled with ones

```
np.ones(5)  
>> array([1., 1., 1., 1., 1.])
```



Creating Arrays 2

Creating an array of evenly spaced values (start, stop, step)

```
np.arange(2,18,4)  
>> array([ 2,  6, 10, 14])
```

Creating an array of random values (shown as 2x3 matrix)

```
np.random.random((2,3))  
>> array([[0.99800537, 0.65104252, 0.15230364],  
         [0.25347108, 0.85345208, 0.44232692]])
```

Creating an array filled with given value (dimensions, value)

```
np.full(4, 10)  
>> array([10, 10, 10, 10])
```

Creating an empty array (fills with arbitrary, un-initialized values)

```
np.empty(2)  
>> array([1.05116502e-311, 0.000000000e+000])
```

Creating Arrays 3

Creating an array of evenly spaced values (low, high, num-values)

```
np.linspace(1,3,5)  
>> array([1. , 1.5, 2. , 2.5, 3. ])
```

Creating an array of random integers (low, high, size)

```
np.random.randint(2,7,5)  
>> array([3, 3, 4, 4, 6])
```


Summary Operations I

Example 1-Dimensional Array

```
my_1d_array = np.random.randint(0,10,7)
>> array([7, 6, 4, 2, 9, 6, 7])
```

Max, Min, Mean, Sum, Standard Deviation

```
my_1d_array.max()
>> 9

my_1d_array.min()
>> 2

my_1d_array.mean()
>> 5.857142857142857

my_1d_array.sum()
>> 41

my_1d_array.std()
>> 2.099562636671296
```

Summary Operations 2

Example 2-Dimensional Array

```
my_2d_array = np.random.randint(0,10,(2,3))  
>> array([[6, 1, 7],  
          [9, 0, 6]])
```

Maximum Value (of all values in array)

```
my_2d_array.max()  
>> 9
```

Maximum Value (of the values in "column")

```
my_2d_array.max(axis = 0)  
>> array([9, 1, 7])
```

Maximum Value (of the values in "row")

```
my_2d_array.max(axis = 1)  
>> array([7, 9])
```

Math Operations I

Example 1-Dimensional Array

```
a = np.array([1,2,3,4,5])  
>> array([1, 2, 3, 4, 5])
```

Addition, Subtraction, Multiplication, Division

```
a + 10  
>> array([11, 12, 13, 14, 15])  
  
a - 10  
>> array([-9, -8, -7, -6, -5])  
  
a * 10  
>> array([10, 20, 30, 40, 50])  
  
a / 10  
>> array([0.1, 0.2, 0.3, 0.4, 0.5])
```

Math Operations 2

Example 1-Dimensional Array

```
a = np.array([-2,-1,0,1,2])
```

```
>> array([-2, -1,  0,  1,  2])
```

Square, Square Root, Trigonometry, Signs (Positive or Negative)

```
np.square(a)
```

```
>> array([4, 1, 0, 1, 4])
```

```
np.sqrt(a)
```

```
>> array([nan, nan, 0. , 1. , 1.41421356])
```

```
np.sin(a)
```

```
>> array([-0.9092, -0.8414,  0. ,  0.8414,  0.9092])
```

```
np.cos(a)
```

```
>> array([-0.4161,  0.5403,  1. ,  0.5403, -0.4161])
```

```
np.tan(a)
```

```
>> array([ 2.1850, -1.5574,  0. ,  1.5574, -2.1850])
```

```
np.sign(a)
```

```
>> array([-1, -1,  0,  1,  1])
```

Math Operations 3

Example: Multiple Arrays

```
a = np.array([1,2,3])  
b = np.array([4,5,6])
```

Arithmetic on multiple arrays & dot product

```
np.add(a,b) # or a + b  
>> array([5, 7, 9])  
  
np.subtract(a,b) # or a - b  
>> array([-3, -3, -3])  
  
np.multiply(a,b) # or a * b  
>> array([ 4, 10, 18])  
  
np.divide(a,b) # or a / b  
>> array([0.25, 0.4 , 0.5 ])  
  
np.dot(a,b)  
>> 32
```

Array Comparison

Example: Multiple Arrays

```
a = np.array([1,2,3])  
b = np.array([4,2,6])
```

Element-wise comparison (which elements are equal)

```
a == b  
>> array([False,  True, False])
```

Array comparison (are arrays equal)

```
np.array_equal(a, b)  
>> False
```

Accessing Elements I

Example 1-Dimensional Array

```
a = np.array([1,2,3,4,5])  
>> array([1, 2, 3, 4, 5])
```

Element at specific index

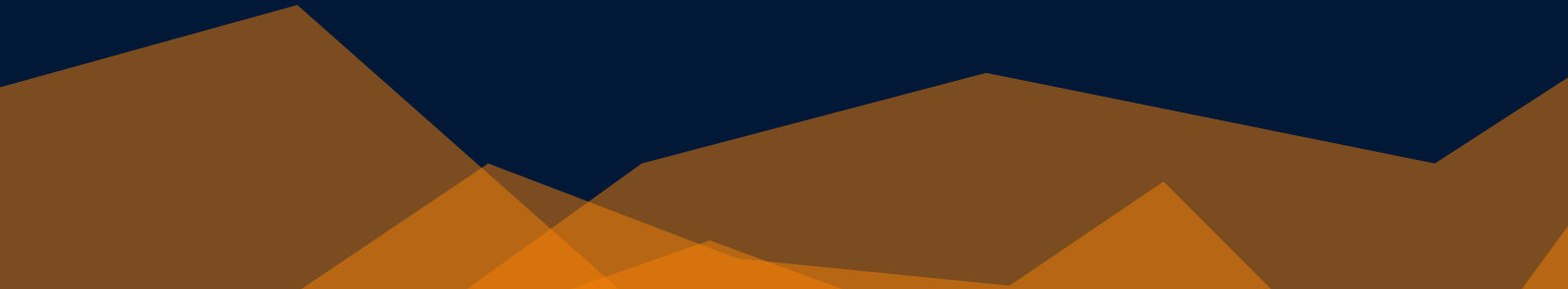
```
a[0]  
>> 1
```

Slicing

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

Last Element

```
a[-1]  
>> 5
```



Accessing Elements 2

Example 2-Dimensional Array

```
a = np.array([[1,2,3],[4,5,6]])  
>> array([[1, 2, 3],  
          [4, 5, 6]])
```

Element at specific index (here this returns "row" 1)

```
a[0]  
>> array([1, 2, 3])
```

Element at specific index (here this returns "row" 1, "column" 2)

```
a[0][1]  
>> 2
```


Re-shaping Arrays

Example 2-Dimensional Array

```
a = np.array([[1,2,3],[4,5,6]])  
>> array([[1, 2, 3],  
          [4, 5, 6]])
```

Re-shape (here to 3 rows & 2 columns)

```
a.reshape(3,2)  
>> array([[1, 2],  
          [3, 4],  
          [5, 6]])
```

Re-shape (here to a 1-Dimensional array)

```
a.flatten() # or a.reshape(6) or a.ravel()  
>> array([1, 2, 3, 4, 5, 6])
```

Stacking Arrays

Example: Multiple Arrays

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

```
>> array([1, 2, 3])
```

```
b = np.array([4,5,6])
```

```
>> array([4, 5, 6])
```

Horizontal Stack

```
np.hstack((a,b))
```

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

Vertical Stack

```
np.vstack((a,b))
```

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

Array Features

Example 2-Dimensional Array

```
a = np.array([[1,2,3],[4,5,6]])  
>> array([[1, 2, 3],  
          [4, 5, 6]])
```

Array Shape (length of each dimension)

```
a.shape  
>> (2, 3)
```

Number of dimensions

```
a.ndim  
>> 2
```

Number of elements

```
a.size  
>> 6
```

A Planetary Example!

Here we are going to calculate the **volumes** of the eight planets in our solar system - based upon their **radius measurements**.

We are going to do this all using **NumPy!**




After that - we're going to crank it up! Instead of just doing this for eight planets, we will run this for **one million** made-up planets.

This will not only showcase some of the key functionality of NumPy - but also the incredible **speed** at which it can run calculations!

Our data...

We have a NumPy array holding the radius distance for each of the eight planets!

Planet	Radius (km)
Mercury	2439.7
Venus	6051.8
Earth	6371
Mars	3389.7
Jupiter	69911
Saturn	58232
Uranus	25362
Neptune	24622



```
radii = np.array([2439.7, 6051.8, 6371, 3389.7, 69911, 58232, 25362, 24622])
```

```
>> array([ 2439.7, 6051.8, 6371., 3389.7, 69911., 58232., 25362., 24622.])
```

Calculating the volumes!

Here we will create a new array called **volumes** and we will apply the formula for calculating volume from radius to our **radii** array!

Formula

$$\text{volume (sphere)} = \frac{4}{3} \pi r^3$$

radius

```
volumes = 4/3 * np.pi * radii**3
```

```
>> [6.08e+10 9.28e+11 1.08e+12 1.63e+11, 1.43e+15 8.27e+14 6.83e+13 6.25e+13]
```

Because of the way NumPy works - it does this very, very quickly

It applies the volume formula to each of the values of the radii array essentially in one go, rather than looping through them one at a time, meaning the overall time taken is low.

Eight is easy though - let's crank it up a notch...

Cranking it up...

Instead of the radius measurements for eight planets, we will create an array that contains measurements for one million made-up planets. To do this we use the **random** functionality within NumPy to ask for one million random integers, each which will be a value between 1 and 1,000...

```
radii = np.random.randint(1, 1000, 1000000)
```

Since we have over-written the **radii** array, we can run the same volume calculation code below...

Before you hit run - how long do you think it will take to run this calculation for **one million** planets?!

```
volumes = 4/3 * np.pi * radii**3
```

Run! **Wow, NumPy is faaast!**

It processed this for one million planets, in a fraction of a second! This is a simple example of why this can be such a useful library for tasks in Python!

