Numpy

NumPy (or Numpy) is a Linear Algebra Library for Python, the reason it is so important for Data Science with Python is that almost all of the libraries in the Python Data Ecosystem rely on NumPy as one of their main building blocks.

```
In [1]: import numpy as np
```

Python List can be converted into numpy array

```
In [3]: my_list = [1,2,3]
        print(my_list)
```

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

```
In [4]: np.array(my_list)
```

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

In-Built Functions

arange

Return evenly spaced values within a given interval.

```
In [5]: np.arange(0,10)
```

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

```
In [6]: np.arange(0,10,2)
```

```
Out[6]: array([0, 2, 4, 6, 8])
```

zeros and ones

Generate arrays of zeros or ones

```
In [8]: np.zeros(5)
```

```
Out[8]: array([0., 0., 0., 0., 0.])
```

```
In [11]: np.zeros((5,5))
```

```
Out[11]: array([[0., 0., 0., 0., 0.],
                [0., 0., 0., 0., 0.],
                [0., 0., 0., 0., 0.],
                [0., 0., 0., 0., 0.],
                [0., 0., 0., 0., 0.]])
```

```
In [12]: np.ones(4)
```

```
Out[12]: array([1., 1., 1., 1.])
```

linspace

Return evenly spaced numbers over a specified interval.

```
In [13]: np.linspace(0,10,3)
```

```
Out[13]: array([ 0.,  5., 10.])
```

```
In [15]: np.linspace(0,5,50)
```

```
Out[15]: array([0.        , 0.10204082, 0.20408163, 0.30612245, 0.40816327,
                0.51020408, 0.6122449 , 0.71428571, 0.81632653, 0.91836735,
                1.02040816, 1.12244898, 1.2244898 , 1.32653061, 1.42857143,
                1.53061224, 1.63265306, 1.73469388, 1.83673469, 1.93877551,
                2.04081633, 2.14285714, 2.24489796, 2.34693878, 2.44897959,
                2.55102041, 2.65306122, 2.75510204, 2.85714286, 2.95918367,
                3.06122449, 3.16326531, 3.26530612, 3.36734694, 3.46938776,
                3.57142857, 3.67346939, 3.7755102 , 3.87755102, 3.97959184,
                4.08163265, 4.18367347, 4.28571429, 4.3877551 , 4.48979592,
                4.59183673, 4.69387755, 4.79591837, 4.89795918, 5.        ])
```

eye

Creates an identity matrix

```
In [16]: np.eye(5)
```

```
Out[16]: array([[1., 0., 0., 0., 0.],
                [0., 1., 0., 0., 0.],
                [0., 0., 1., 0., 0.],
                [0., 0., 0., 1., 0.],
                [0., 0., 0., 0., 1.]])
```

Random

Numpy also has lots of ways to create random number arrays:

rand

Create an array of the given shape and populate it with random samples from a uniform distribution over `[0, 1)`.

```
In [19]: np.random.rand(3)
```

```
Out[19]: array([0.17150528, 0.57919722, 0.0327859 ])
```

```
In [18]: np.random.rand(2,3)
```

```
Out[18]: array([[0.85141086, 0.3996946 , 0.44741098],
                [0.23999596, 0.28645291, 0.56059505]])
```

randn

Return a sample (or samples) from the "standard normal" distribution. Unlike rand which is uniform:

In [20]:
```
np.random.randn(3)
```

Out[20]:
```
array([ 1.60635037, -0.02386401,  1.3855219 ])
```

In [21]:
```
np.random.randn(2,3)
```

Out[21]:
```
array([[ 0.59951302, -1.10276527,  0.03509468],
       [-0.50159196,  0.07893611,  0.77122671]])
```

randint

Return random integers from `low` (inclusive) to `high` (exclusive).

In [22]:
```
np.random.randint(1,50)
```

Out[22]:
```
14
```

In [23]:
```
np.random.randint(1,50,5)
```

Out[23]:
```
array([28,  4, 15, 11, 28])
```

Array Attributes and Methods

In [26]:
```
arr = np.arange(15)
arr
```

Out[26]:
```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14])
```

Reshape

Returns an array containing the same data with a new shape.

In [25]:
```
arr.reshape(3,5)
```

Out[25]:
```
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14]])
```

max,min,argmax,argmin

These are useful methods for finding max or min values. Or to find their index locations using argmin or argmax

In [29]:
```
ran = np.random.randint(1,50,10)
ran
```

Out[29]:
```
array([36, 42, 27, 45, 12, 29,  5, 32, 12, 17])
```

In [30]:
```
ran.max()
```

Out[30]:
```
45
```

In [31]:
```
ran.min()
```

Out[31]:
```
5
```

```
In [32]: ran.argmax()
Out[32]: 3

In [33]: ran.argmin()
Out[33]: 6
```

Shape

Shape is an attribute that arrays have

```
In [35]: arr.shape
Out[35]: (15,)

In [36]: arr.reshape(1,15)
Out[36]: array([[ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14]])

In [38]: arr.reshape(1,15).shape
Out[38]: (1, 15)
```

dtype

You can also grab the data type of the object in the array:

```
In [39]: arr.dtype
Out[39]: dtype('int64')

In [41]: arr=np.arange(10)
```

Bracket Indexing and Selection

The simplest way to pick one or some elements of an array looks very similar to python lists:

```
In [42]: arr[1:3]
Out[42]: array([1, 2])
```

Broadcasting

Numpy arrays differ from a normal Python list because of their ability to broadcast:

```
In [44]: arr[1:3]=20
         arr
Out[44]: array([ 0, 20, 20,  3,  4,  5,  6,  7,  8,  9])

In [46]: arr = np.arange(1,11)
         arr
Out[46]: array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10])
```

```
In [48]: slice_arr = arr[0:4]
         slice_arr
```

```
Out[48]: array([1, 2, 3, 4])
```

```
In [49]: slice_arr[:]=20
```

```
In [50]: arr
```

```
Out[50]: array([20, 20, 20, 20,  5,  6,  7,  8,  9, 10])
```

Change get reflected on the original array. Data is not copied, it's a view of the original array! This avoids memory problems!

Selection

Let's briefly go over how to use brackets for selection based off of comparison operators.

```
In [53]: arr = np.arange(1,11)
         arr
```

```
Out[53]: array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10])
```

```
In [54]: arr > 5
```

```
Out[54]: array([False, False, False, False, False,  True,  True,  True,  True,
                 True])
```

```
In [56]: bool_arr = arr >5
         bool_arr
```

```
Out[56]: array([False, False, False, False, False,  True,  True,  True,  True,
                 True])
```

```
In [57]: arr[bool_arr]
```

```
Out[57]: array([ 6,  7,  8,  9, 10])
```

```
In [59]: arr[arr>7]
```

```
Out[59]: array([ 8,  9, 10])
```

NumPy Operations

Arithmetic

You can easily perform array with array arithmetic, or scalar with array arithmetic. Let's see some examples:

```
In [61]: arr = np.arange(1,11)
         arr
```

```
Out[61]: array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10])
```

```
In [62]: arr + arr
```

```
Out[62]: array([ 2,  4,  6,  8, 10, 12, 14, 16, 18, 20])
```

```
In [63]: arr * arr
```

```
Out[63]: array([  1,   4,   9,  16,  25,  36,  49,  64,  81, 100])
```

```
In [64]: arr / arr
```

```
Out[64]: array([1., 1., 1., 1., 1., 1., 1., 1., 1., 1.])
```

```
In [65]: 1/arr
```

```
Out[65]: array([1.        , 0.5       , 0.33333333, 0.25      , 0.2       ,
                0.16666667, 0.14285714, 0.125     , 0.11111111, 0.1       ])
```

```
In [66]: arr - arr
```

```
Out[66]: array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
```

```
In [67]: arr ** 3
```

```
Out[67]: array([   1,    8,   27,   64,  125,  216,  343,  512,  729, 1000])
```

```
In [68]: np.sqrt(arr)
```

```
Out[68]: array([1.        , 1.41421356, 1.73205081, 2.        , 2.23606798,
                2.44948974, 2.64575131, 2.82842712, 3.        , 3.16227766])
```

```
In [69]: np.exp(arr)
```

```
Out[69]: array([2.71828183e+00, 7.38905610e+00, 2.00855369e+01, 5.45981500e+01,
                1.48413159e+02, 4.03428793e+02, 1.09663316e+03, 2.98095799e+03,
                8.10308393e+03, 2.20264658e+04])
```

```
In [70]: np.max(arr)
```

```
Out[70]: 10
```

```
In [71]: np.sin(arr)
```

```
Out[71]: array([ 0.84147098,  0.90929743,  0.14112001, -0.7568025 , -0.95892427,
                -0.2794155 ,  0.6569866 ,  0.98935825,  0.41211849, -0.54402111])
```

```
In [72]: np.log(arr)
```

```
Out[72]: array([0.        , 0.69314718, 1.09861229, 1.38629436, 1.60943791,
                1.79175947, 1.94591015, 2.07944154, 2.19722458, 2.30258509])
```

Data Types

bool_ Boolean (True or False) stored as a byte

int_ Default integer type (same as C long; normally either int64 or int32)

intc Identical to C int (normally int32 or int64)

intp Integer used for indexing (same as C ssize_t; normally either int32 or int64)

int8 Byte (-128 to 127)

int16 Integer (-32768 to 32767)

int32 Integer (-2147483648 to 2147483647)

int64 Integer (-9223372036854775808 to 9223372036854775807)

uint8 Unsigned integer (0 to 255)

uint16 Unsigned integer (0 to 65535)

uint32 Unsigned integer (0 to 4294967295)

uint64 Unsigned integer (0 to 18446744073709551615)

float_ Shorthand for float64

float16 Half precision float: sign bit, 5 bits exponent, 10 bits mantissa

float32 Single precision float: sign bit, 8 bits exponent, 23 bits mantissa

float64 Double precision float: sign bit, 11 bits exponent, 52 bits mantissa

complex_ Shorthand for complex128

complex64 Complex number, represented by two 32-bit floats (real and imaginary components)

complex128 Complex number, represented by two 64-bit floats (real and imaginary components)