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 PyData Ecosystem rely on NumPy as one of their main building blocks.

Numpy is also incredibly fast, as it has bindings to C libraries. For more info on why you would want to use Arrays instead of lists, check out this great StackOverflow post (http://stackoverflow.com/questions/993984/why-numpy-instead-of-python-lists).

We will install the NumPy modeule and learn the basics of NumPy.

Installation Instructions

I hope you have successfully installed Python using Anaconda distribution. After installing Anaconda, to install NumPy, go to your terminal or command prompt and type:

```
conda install numpy
```

Using NumPy

Once you've installed NumPy you can import it as a library:

In [3]:
```
import numpy as np
```

Numpy has many built-in functions and capabilities. We will focus on some of the most important aspects of Numpy: vectors,arrays,matrices, and number generation. Let's start by discussing arrays.

Numpy Arrays

Numpy arrays essentially come in two flavors: vectors and matrices. Vectors are strictly 1-d arrays and matrices are 2-d (but you should note a matrix can still have only one row or one column).

Let's begin our introduction by exploring how to create NumPy arrays.

Creating NumPy Arrays

From a Python List

We can create an array by directly converting a list or list of lists:

In [4]:
```
my_list = [1,2,3] # create a list
my_list
```

Out[4]:
```
[1, 2, 3]
```

In [5]:
```
np.array(my_list) # convert the list to an array
```

Out[5]:
```
array([1, 2, 3])
```

In [6]:
```
my_matrix = [[1,2,3],[4,5,6],[7,8,9]]  # create a matrix
my_matrix
```

Out[6]:
```
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

In [7]:
```
np.array(my_matrix) # convert the matrix in an array
```

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

Built-in Methods

There are lots of built-in ways to generate Arrays

arange

Return evenly spaced values within a given interval.

```
In [8]: np.arange(0,10)
```

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

```
In [9]: np.arange(0,11,2)
```

```
Out[9]: array([ 0,  2,  4,  6,  8, 10])
```

zeros and ones

Generate arrays of zeros or ones

```
In [10]: np.zeros(3)
```

```
Out[10]: array([ 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(3)
```

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

```
In [13]: np.ones((3,3))
```

```
Out[13]: array([[ 1.,  1.,  1.],
                [ 1.,  1.,  1.],
                [ 1.,  1.,  1.]])
```

linspace

Return evenly spaced numbers over a specified interval.

```
In [14]: np.linspace(0,10,3)
```

```
Out[14]: array([  0.,   5.,  10.])
```

```
In [15]: np.linspace(0,10,50)
```

```
Out[15]: array([  0.        ,   0.20408163,   0.40816327,   0.6122449 ,
                  0.81632653,   1.02040816,   1.2244898 ,   1.42857143,
                  1.63265306,   1.83673469,   2.04081633,   2.24489796,
                  2.44897959,   2.65306122,   2.85714286,   3.06122449,
                  3.26530612,   3.46938776,   3.67346939,   3.87755102,
                  4.08163265,   4.28571429,   4.48979592,   4.69387755,
                  4.89795918,   5.10204082,   5.30612245,   5.51020408,
                  5.71428571,   5.91836735,   6.12244898,   6.32653061,
                  6.53061224,   6.73469388,   6.93877551,   7.14285714,
                  7.34693878,   7.55102041,   7.75510204,   7.95918367,
                  8.16326531,   8.36734694,   8.57142857,   8.7755102 ,
                  8.97959184,   9.18367347,   9.3877551 ,   9.59183673,
                  9.79591837,  10.        ])
```

eye

Creates an identity matrix

```
In [16]: np.eye(4)
```

```
Out[16]: array([[ 1.,  0.,  0.,  0.],
                [ 0.,  1.,  0.,  0.],
                [ 0.,  0.,  1.,  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 [17]: np.random.rand(2)
```

```
Out[17]: array([ 0.71306692,  0.64316647])
```

```
In [18]: np.random.rand(5,5)
```

```
Out[18]: array([[ 0.57768353,  0.17287132,  0.02243031,  0.57308228,  0.33228228],
                [ 0.41556544,  0.33795959,  0.03239384,  0.15163535,  0.01003505],
                [ 0.01288373,  0.4640226 ,  0.76730903,  0.47233347,  0.50511256],
                [ 0.00579912,  0.14735287,  0.0729522 ,  0.24603117,  0.66389827],
                [ 0.43255062,  0.7092671 ,  0.3027965 ,  0.52162253,  0.46419041]])
```

randn

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

```
In [19]: np.random.randn(2)
```

```
Out[19]: array([ 1.66700486, -0.4654533 ])
```

```
In [20]: np.random.randn(5,5)
```

```
Out[20]: array([[ 0.66561656, -0.88539326, -1.31192316,  0.68184587, -0.42229124],
                [ 0.20450859,  1.95557129,  0.62273082, -0.7600267 , -0.85334959],
                [ 0.13715454,  1.21880774,  0.23617308,  0.39340878, -3.81382669],
                [ 1.18653905,  0.1668961 ,  1.45330127, -1.25887507, -0.46755675],
                [ 2.19618744, -1.51501549,  0.51127346,  0.6909019 ,  0.32373068]])
```

randint

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

```
In [21]: np.random.randint(1,100)
```

```
Out[21]: 13
```

```
In [22]: np.random.randint(1,100,10)
```

```
Out[22]: array([57, 16, 29, 41, 58, 66, 18, 70, 19, 87])
```

Array Attributes and Methods

Let's discuss some useful attributes and methods or an array:

```
In [23]: arr = np.arange(25)
         ranarr = np.random.randint(0,50,10)
```

```
In [24]: arr
```

```
Out[24]: array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16,
                17, 18, 19, 20, 21, 22, 23, 24])
```

```
In [25]: ranarr
```

```
Out[25]: array([38,  3,  2, 47, 22,  9, 15, 29, 35, 23])
```

Reshape

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

```
In [26]: arr.reshape(5,5)
```

```
Out[26]: array([[ 0,  1,  2,  3,  4],
                [ 5,  6,  7,  8,  9],
                [10, 11, 12, 13, 14],
                [15, 16, 17, 18, 19],
                [20, 21, 22, 23, 24]])
```

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 [27]: ranarr
```

```
Out[27]: array([38,  3,  2, 47, 22,  9, 15, 29, 35, 23])
```

```
In [28]: ranarr.max()
```

```
Out[28]: 47
```

```
In [29]: ranarr.argmax()
```

```
Out[29]: 3
```

```
In [30]: ranarr.min()
```

```
Out[30]: 2
```

```
In [31]: ranarr.argmin()
```

```
Out[31]: 2
```

Shape

Shape is an attribute that arrays have (not a method):

In [32]:
```
# Vector
arr.shape
```

Out[32]:
```
(25,)
```

In [33]:
```
# Notice the two sets of brackets
arr.reshape(1,25)
```

Out[33]:
```
array([[ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16,
        17, 18, 19, 20, 21, 22, 23, 24]])
```

In [34]:
```
arr.reshape(1,25).shape
```

Out[34]:
```
(1, 25)
```

In [35]:
```
arr.reshape(25,1)
```

Out[35]:
```
array([[ 0],
       [ 1],
       [ 2],
       [ 3],
       [ 4],
       [ 5],
       [ 6],
       [ 7],
       [ 8],
       [ 9],
       [10],
       [11],
       [12],
       [13],
       [14],
       [15],
       [16],
       [17],
       [18],
       [19],
       [20],
       [21],
       [22],
       [23],
       [24]])
```

In [36]:
```
arr.reshape(25,1).shape
```

Out[36]:
```
(25, 1)
```

dtype

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

In [37]:
```
arr.dtype
```

Out[37]:
```
dtype('int64')
```

NumPy Operations

Arithmetic

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

In [38]:
```
import numpy as np
arr = np.arange(0,10)
```

In [39]:
```
arr + arr
```

Out[39]:
```
array([ 0,  2,  4,  6,  8, 10, 12, 14, 16, 18])
```

In [40]:
```
arr * arr
```

Out[40]:
```
array([ 0,  1,  4,  9, 16, 25, 36, 49, 64, 81])
```

In [41]:
```
arr - arr
```

Out[41]:
```
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
```

In [42]:
```
# Warning on division by zero, but not an error!
# Just replaced with nan
arr/arr
```

```
/anaconda/lib/python3.6/site-packages/ipykernel/__main__.py:3: RuntimeWarning: invalid value encountered in true_divide
  app.launch_new_instance()
```

Out[42]:
```
array([ nan,   1.,   1.,   1.,   1.,   1.,   1.,   1.,   1.,   1.])
```

In [43]:
```
# Also warning, but not an error instead infinity
1/arr
```

```
/anaconda/lib/python3.6/site-packages/ipykernel/__main__.py:2: RuntimeWarning: divide by zero encountered in true_divide
  from ipykernel import kernelapp as app
```

Out[43]:
```
array([        inf,  1.        ,  0.5       ,  0.33333333,  0.25      ,
        0.2       ,  0.16666667,  0.14285714,  0.125     ,  0.11111111])
```

In [44]:
```
arr**3
```

Out[44]:
```
array([  0,   1,   8,  27,  64, 125, 216, 343, 512, 729])
```

Universal Array Functions

Numpy comes with many universal array functions (http://docs.scipy.org/doc/numpy/reference/ufuncs.html), which are essentially just mathematical operations you can use to perform the operation across the array. Let's see some common ones:

In [45]:
```
#Taking Square Roots
np.sqrt(arr)
```

Out[45]:
```
array([ 0.        ,  1.        ,  1.41421356,  1.73205081,  2.        ,
        2.23606798,  2.44948974,  2.64575131,  2.82842712,  3.        ])
```

In [46]:
```
#Calcualting exponential (e^)
np.exp(arr)
```

Out[46]:
```
array([  1.00000000e+00,   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])
```

In [47]:
```
np.max(arr) #same as arr.max()
```

Out[47]:
```
9
```

In [48]:
```
np.sin(arr)
```

Out[48]:
```
array([ 0.        ,  0.84147098,  0.90929743,  0.14112001, -0.7568025 ,
       -0.95892427, -0.2794155 ,  0.6569866 ,  0.98935825,  0.41211849])
```

In [49]:
```
np.log(arr)
```

```
/anaconda/lib/python3.6/site-packages/ipykernel/__main__.py:1: RuntimeWarning: divide by zero encountered in log
  if __name__ == '__main__':
```

Out[49]:
```
array([       -inf,  0.        ,  0.69314718,  1.09861229,  1.38629436,
        1.60943791,  1.79175947,  1.94591015,  2.07944154,  2.19722458])
```

NumPy Indexing and Selection

In this section we will discuss how to select elements or groups of elements from an array.

In [50]:
```
import numpy as np
```

In [51]:
```
#Creating sample array
arr = np.arange(0,11)
```

In [52]:
```
#Show
arr
```

Out[52]:
```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10])
```

Bracket Indexing and Selection

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

In [53]:
```
#Get a value at an index
arr[8]
```

Out[53]:
```
8
```

In [54]:
```
#Get values in a range
arr[1:5]
```

Out[54]: array([1, 2, 3, 4])

In [55]:
```
#Get values in a range
arr[0:5]
```

Out[55]: array([0, 1, 2, 3, 4])

Broadcasting

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

In [56]:
```
#Setting a value with index range (Broadcasting)
arr[0:5]=100

#Show
arr
```

Out[56]: array([100, 100, 100, 100, 100,   5,   6,   7,   8,   9, 10])

In [57]:
```
# Reset array, we'll see why I had to reset in  a moment
arr = np.arange(0,11)

#Show
arr
```

Out[57]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10])

In [58]:
```
#Important notes on Slices
slice_of_arr = arr[0:6]

#Show slice
slice_of_arr
```

Out[58]: array([0, 1, 2, 3, 4, 5])

In [59]:
```
#Change Slice
slice_of_arr[:]=99

#Show Slice again
slice_of_arr
```

Out[59]:
```
array([99, 99, 99, 99, 99, 99])
```

Now note the changes also occur in our original array!

In [61]:
```
arr
```

Out[61]:
```
array([99, 99, 99, 99, 99, 99,  6,  7,  8,  9, 10])
```

Data is not copied, it's a view of the original array! This avoids memory problems!

In [62]:
```
#To get a copy, need to be explicit
arr_copy = arr.copy()

arr_copy
```

Out[62]:
```
array([99, 99, 99, 99, 99, 99,  6,  7,  8,  9, 10])
```

Indexing a 2D array (matrices)

The general format is **arr_2d[row][col]** or **arr_2d[row,col]**. I recommend usually using the comma notation for clarity.

In [63]:
```
arr_2d = np.array(([5,10,15],[20,25,30],[35,40,45]))

#Show
arr_2d
```

Out[63]:
```
array([[ 5, 10, 15],
       [20, 25, 30],
       [35, 40, 45]])
```

In [65]:
```
#Indexing row
arr_2d[1]
```

Out[65]:
```
array([20, 25, 30])
```

In [66]:
```
# Format is arr_2d[row][col] or arr_2d[row,col]

# Getting individual element value
arr_2d[1][0]
```

Out[66]:
```
20
```

In [67]:
```
# Getting individual element value
arr_2d[1,0]
```

Out[67]:
```
20
```

In [68]:
```
# 2D array slicing

#Shape (2,2) from top right corner
arr_2d[:2,1:]
```

Out[68]:
```
array([[10, 15],
       [25, 30]])
```

In [69]:
```
#Shape bottom row
arr_2d[2]
```

Out[69]:
```
array([35, 40, 45])
```

In [70]:
```
#Shape bottom row
arr_2d[2,:]
```

Out[70]:
```
array([35, 40, 45])
```

Selection

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

In [71]:
```
arr = np.arange(1,11)
arr
```

Out[71]: array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])

In [72]:
```
arr > 4
```

Out[72]: array([False, False, False, False, True, True, True, True, True, True], dtype=bool)

In [73]:
```
bool_arr = arr>4
bool_arr
```

Out[73]: array([False, False, False, False, True, True, True, True, True, True], dtype=bool)

In [74]:
```
arr[bool_arr]
```

Out[74]: array([5, 6, 7, 8, 9, 10])

In [75]:
```
arr[arr>2]
```

Out[75]: array([3, 4, 5, 6, 7, 8, 9, 10])

In [76]:
```
x = 2
arr[arr>x]
```

Out[76]: array([3, 4, 5, 6, 7, 8, 9, 10])

NumPy More Examples

You can go through the following examples for more practice

Create an array of 10 zeros

In [77]:
```
np.zeros(10)
```

Out[77]: array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])

Create an array of 10 ones

```
In [79]: np.ones(10)
```

```
Out[79]: array([ 1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.])
```

Create an array of 10 fives

np.ones(10) * 5

Create an array of the integers from 10 to 50

```
In [81]: np.arange(10,51)
```

```
Out[81]: array([10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26,
                27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43,
                44, 45, 46, 47, 48, 49, 50])
```

Create an array of all the even integers from 10 to 50

```
In [82]: np.arange(10,51,2)
```

```
Out[82]: array([10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42,
                44, 46, 48, 50])
```

Create a 3x3 matrix with values ranging from 0 to 8

```
In [83]: np.arange(9).reshape(3,3)
```

```
Out[83]: array([[0, 1, 2],
                [3, 4, 5],
                [6, 7, 8]])
```

Create a 3x3 identity matrix

```
In [84]: np.eye(3)
```

```
Out[84]: array([[ 1.,  0.,  0.],
                [ 0.,  1.,  0.],
                [ 0.,  0.,  1.]])
```

Use NumPy to generate a random number between 0 and 1

```
In [85]: np.random.rand(1)
```

```
Out[85]: array([ 0.26877465])
```

Use NumPy to generate an array of 25 random numbers sampled from a standard normal distribution

```
In [86]: np.random.randn(25)
```

```
Out[86]: array([-1.52957054, -0.41073193, -1.68350777, -0.0276512 ,  0.3377702 ,
                -0.51048213, -1.42902227,  0.36214364, -0.00255689, -0.71222602,
                -0.36555485,  1.18618604, -0.28126423,  0.43800053, -1.73977583,
                 0.13028251, -0.69230744,  0.04555812, -1.49207309,  0.23017301,
                 0.05984923, -0.51013499, -0.39348911,  0.2076578 ,  0.41894075])
```

Create the following matrix:

```
In [87]: np.arange(1,101).reshape(10,10) / 100
```

```
Out[87]: array([[ 0.01,  0.02,  0.03,  0.04,  0.05,  0.06,  0.07,  0.08,  0.09,  0.1 ],
                [ 0.11,  0.12,  0.13,  0.14,  0.15,  0.16,  0.17,  0.18,  0.19,  0.2 ],
                [ 0.21,  0.22,  0.23,  0.24,  0.25,  0.26,  0.27,  0.28,  0.29,  0.3 ],
                [ 0.31,  0.32,  0.33,  0.34,  0.35,  0.36,  0.37,  0.38,  0.39,  0.4 ],
                [ 0.41,  0.42,  0.43,  0.44,  0.45,  0.46,  0.47,  0.48,  0.49,  0.5 ],
                [ 0.51,  0.52,  0.53,  0.54,  0.55,  0.56,  0.57,  0.58,  0.59,  0.6 ],
                [ 0.61,  0.62,  0.63,  0.64,  0.65,  0.66,  0.67,  0.68,  0.69,  0.7 ],
                [ 0.71,  0.72,  0.73,  0.74,  0.75,  0.76,  0.77,  0.78,  0.79,  0.8 ],
                [ 0.81,  0.82,  0.83,  0.84,  0.85,  0.86,  0.87,  0.88,  0.89,  0.9 ],
                [ 0.91,  0.92,  0.93,  0.94,  0.95,  0.96,  0.97,  0.98,  0.99,  1.  ]])
```

Create an array of 20 linearly spaced points between 0 and 1:

```
In [88]: np.linspace(0,1,20)
```

```
Out[88]: array([ 0.        ,  0.05263158,  0.10526316,  0.15789474,  0.21052632,
                 0.26315789,  0.31578947,  0.36842105,  0.42105263,  0.47368421,
                 0.52631579,  0.57894737,  0.63157895,  0.68421053,  0.73684211,
                 0.78947368,  0.84210526,  0.89473684,  0.94736842,  1.        ])
```

Some additional useful functions

```
In [89]: mat = np.arange(1,26).reshape(5,5)
         mat
```

```
Out[89]: array([[ 1,  2,  3,  4,  5],
                [ 6,  7,  8,  9, 10],
                [11, 12, 13, 14, 15],
                [16, 17, 18, 19, 20],
                [21, 22, 23, 24, 25]])
```

Get the sum of all the values in mat

```
In [90]: mat.sum()
Out[90]: 325
```

Get the standard deviation of the values in mat

```
In [91]: mat.std()
Out[91]: 7.2111025509279782
```

Get the sum of all the columns in mat

```
In [92]: mat.sum(axis=0)
Out[92]: array([55, 60, 65, 70, 75])
```

Get the sum of all the rows in mat

```
In [93]: mat.sum(axis=1)
Out[93]: array([ 15,  40,  65,  90, 115])
```

I hope you enjoyed learning NumPy and some of its most useful functions and operations. If you want to explore NumPy even further I suggest you visit the NumPy documentation page (https://docs.scipy.org/doc/numpy/reference/)

Great Job!