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.

Installation Instructions

It is highly recommended you install Python using the Anaconda distribution to make sure all underlying dependencies (such as Linear Algebra libraries) all sync up with the use of a conda install. If you have Anaconda, install NumPy by going to your terminal or command prompt and typing:

```
conda install numpy
```

In [ ]:

```
pip install numpy

install pip
```

In [1]:

```
1 import numpy as np
```

Numpy has many built-in functions and capabilities. Important aspects of Numpy: vectors,arrays,matrices, and number generation.

Numpy Arrays ¶

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).

Creating NumPy Arrays

From a Python List

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

From a Python List

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

In [2]:
```
my_list = [1,2,3]
my_list
```

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

In [3]:
```
np.array(my_list)
```

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

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

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

In [5]:
```
np.array(my_matrix)
```

```
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 [6]:

```
np.arange(0,2)
```

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

In [7]:

```
np.arange(0,12,2)
```

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

In [8]:

```
np.arange(30 , 0 , -3)
```

```
array([30, 27, 24, 21, 18, 15, 12,  9,  6,  3])
```

In [9]:

```
np.arange(0 , 40 ,4)
```

```
array([ 0,  4,  8, 12, 16, 20, 24, 28, 32, 36])
```

Generate arrays of zeros or ones

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

```
array([0., 0., 0.])
```

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

```
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]:
```
1 np.ones(3)
```

```
array([1., 1., 1.])
```

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

```
array([[1., 1., 1.],
       [1., 1., 1.],
       [1., 1., 1.]])
```

linspace

Return evenly spaced numbers over a specified interval.

In [ ]:

```
1
2 np.linspace(0,10,endpoint= False)
```

In [14]:

```
1 np.linspace(0,10,50)
```

```
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 [19]:

```
1 np.eye(8)
```

```
array([[1., 0., 0., 0., 0., 0., 0., 0.],
       [0., 1., 0., 0., 0., 0., 0., 0.],
       [0., 0., 1., 0., 0., 0., 0., 0.],
       [0., 0., 0., 1., 0., 0., 0., 0.],
       [0., 0., 0., 0., 1., 0., 0., 0.],
       [0., 0., 0., 0., 0., 1., 0., 0.],
       [0., 0., 0., 0., 0., 0., 1., 0.],
       [0., 0., 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 [20]:
```
np.random.rand(2)
```

```
array([0.08337239, 0.16083704])
```

In [23]:
```
np.random.rand(2,2,4)
```

```
array([[[0.57859745, 0.06731807, 0.34442599, 0.64136241],
        [0.78861669, 0.90529016, 0.09215032, 0.1338163 ]],

       [[0.6896468 , 0.45000301, 0.98384958, 0.66147109],
        [0.08816332, 0.62134026, 0.22409454, 0.52041809]]])
```

Array Attributes and Methods

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

In [26]:
```
1 arr = np.arange(25)
2 ranarr = np.random.randint(0,50,10)
```

In [27]:
```
1 arr
```

```
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 [28]:
```
1 print(arr.shape)
```

```
(25,)
```

In [28]:
```
1 print(arr.shape)
```

```
(25,)
```

In [ ]:
```
1 ranarr
```

Reshape

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

In [30]:
```
1 print(arr1.shape)
```

```
(5, 5)
```

In [29]:
```
1 arr1 =arr.reshape(5,5)
```

In [33]:

```
print(arr1.shape)
print(arr1)
```

```
(5, 5)
[[ 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]]
```

```
2 ranarr = np.random.randint(0,50,10)
```

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 [34]:
```
1 ranarr
```

```
array([46, 18, 40, 16,  3,  9, 49, 14,  2, 47])
```

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

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

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

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

In [34]:
```
1 ranarr
```

```
array([46, 18, 40, 16,  3,  9, 49, 14,  2, 47])
```

In [35]:
```
1 ranarr.max()
```

```
49
```

In [36]:
```
1 ranarr.argmax()
```

```
6
```

In [37]:
```
1 ranarr.min()
```

```
2
```

In [38]:
```
1 ranarr.argmin()
```

```
8
```

dtype

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

In [45]:

```
1 arr.dtype
```

```
dtype('int32')
```

NumPy Indexing and Selection

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

In [2]:
```
#Creating sample array
arr = np.arange(0,11)
```

In [1]:
```

arr
```

```
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 [4]:
```
#Get a value at an index
arr[8]
```

```
8
```

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

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

In [6]:
```
#Get values in a range
arr[0:5]
```

```
array([0, 1, 2, 3, 4])
```

Broadcasting

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

In [7]:

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

#Show
arr
```

```
[ 0  1  2  3  4  5  6  7  8  9 10]
```

```
array([100, 100, 100, 100, 100,   5,   6,   7,   8,   9,  10])
```

```
4 arr
```

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

In [9]:

```
1 #Important notes on Slices
2 slice_of_arr = arr[0:6]
3
4 #Show slice
5 slice_of_arr
```

```
array([0, 1, 2, 3, 4, 5])
```

In [10]:

```
1 #Change Slice
2 slice_of_arr[:]=99
3
4 #Show Slice again
5 slice_of_arr
```

```
array([99, 99, 99, 99, 99, 99])
```

Now note the changes also occur in our original array!

In [11]:

```
arr
```

```
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 [ ]:

```
#To get a copy, need to be explicit
arr_copy = arr.copy()

arr_copy
```

Indexing a 2D array (matrices)

The general format is **arr_2d[row][col]** or **arr_2d[row,col]**.

In [12]:

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


arr_2d
```

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

In [13]:

```
#Indexing row
arr_2d[1]

```

```
array([20, 25, 30])
```

In [14]:

```
# Format is arr_2d[row][col] or arr_2d[row,col]

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

```
20
```

In [15]:

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

```
20
```

In [ ]:

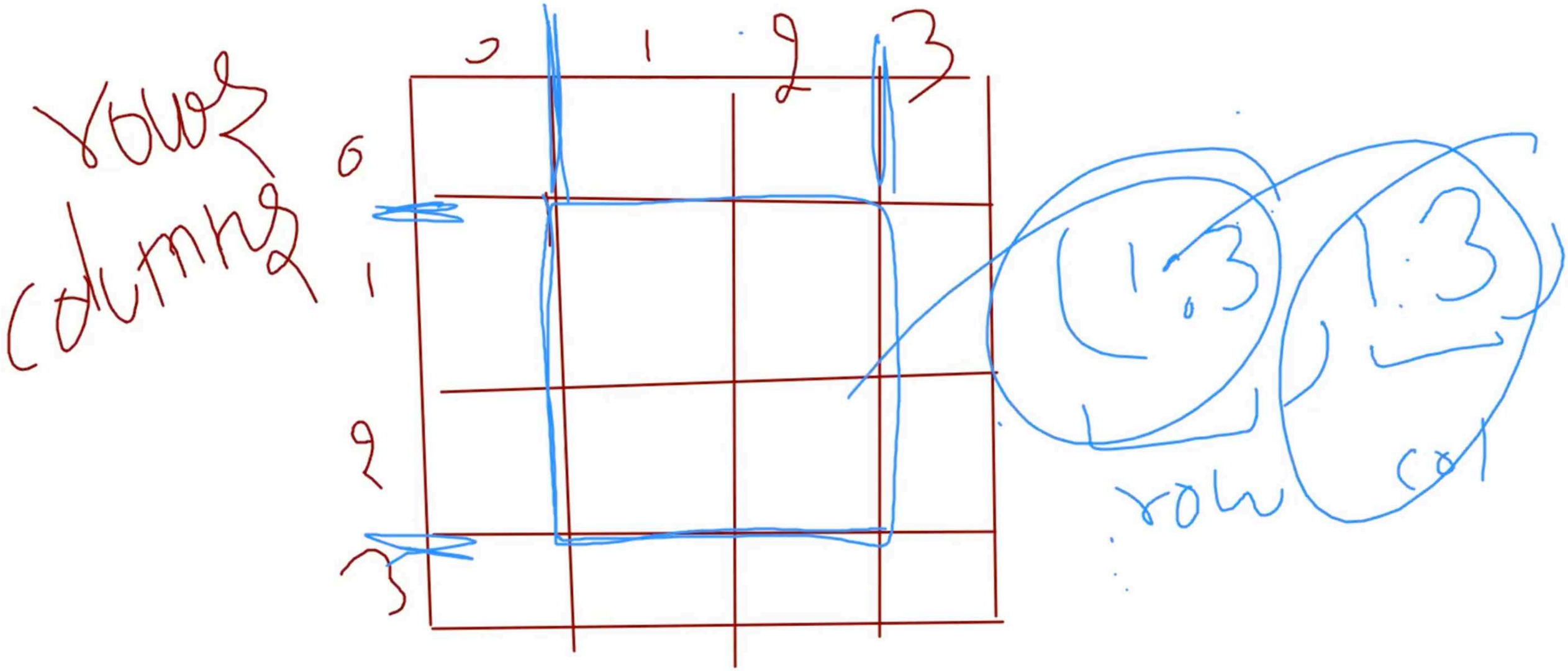
```
# 2D array slicing

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

In [ ]:

```
#Shape bottom row
arr_2d[2]
```

In [ ]:

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


rows
cols
0
1
2
3
0
1
2
(1,3)
(1,3)
row
col

Fancy Indexing

Fancy indexing allows you to select entire rows or columns out of order,to show this, let's quickly build out a numpy array:

In [16]:

```
#Set up matrix
arr2d = np.zeros((10,10))
print(arr2d)
```

```
[[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. 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.]
 [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. 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 [17]:

```
#Length of array
arr_length = arr2d.shape[1]
print(arr_length)
```

```
10
```

In [18]:

```
#Set up array

for i in range(arr_length):
    arr2d[i] = i

arr2d
```

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

Fancy indexing allows the following

In [19]:

```
arr2d[[2,4,6,8]]
```

```
array([[2., 2., 2., 2., 2., 2., 2., 2., 2., 2.],
       [4., 4., 4., 4., 4., 4., 4., 4., 4., 4.],
       [6., 6., 6., 6., 6., 6., 6., 6., 6., 6.],
       [8., 8., 8., 8., 8., 8., 8., 8., 8., 8.]])
```

In [20]:

```
#Allows in any order
arr2d[[6,4,2,7]]
```

```
array([[6., 6., 6., 6., 6., 6., 6., 6., 6., 6.],
       [4., 4., 4., 4., 4., 4., 4., 4., 4., 4.],
       [2., 2., 2., 2., 2., 2., 2., 2., 2., 2.],
       [7., 7., 7., 7., 7., 7., 7., 7., 7., 7.]])
```