Back bone of data type is array.To comprehend array, one must understand dimension first. Sitting in room, one can observe three dimension i.e. x,y & z. Data Dimensions are also three i.e x,y&z.

In [8]:
```
import numpy as np
```

In [9]:
```
#One Dimension array: Moreover, 1D array=vector
a1d = np.array([1,2,3])
```

In [10]:
```
a1d
```

Out[10]:
```
array([1, 2, 3])
```

In [11]:
```
#Two dimension array. Enclosing with an extra square bracket. Moreover, 2D array
a2d = np.array([[1,2,3],[1,2,3],[1,2,3]])
```

In [12]:
```
a2d
```

Out[12]:
```
array([[1, 2, 3],
       [1, 2, 3],
       [1, 2, 3]])
```

In [13]:
```
a3d = np.array([[[1,2,3],[1,2,3],[1,2,3]], [[1,2,3],[1,2,3],[1,2,3]]] )
```

In [14]:
```
a3d
```

Out[14]:
```
array([[[1, 2, 3],
        [1, 2, 3],
        [1, 2, 3]],

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

In [15]:
```
a3d.shape
```

Out[15]:
```
(2, 3, 3)
```

About .shape, Every digit is representing something particular. Moving from right to left: 3:three columns. 3: three rows, & 2:two matrix, of:

In [16]:
```
a2d.shape
```

Out[16]:
```
(3, 3)
```

Here, there are 3 columns and three rows.

In [17]:
```
a1d.shape
```

Out[17]:
```
(3,)
```

Not comprehensible result.

Array is the arrangement of data.

In [18]:
```
#Float is more preffered than array. Look
a2d.dtype
```

Out[18]:
```
dtype('int32')
```

In [19]:
```
#Now changing only first integer into float. Look at the result, it is changed to
ad = np.array([[1.1,2,3],[1,2,3],[1,2,3]])
```

In [20]:
```
ad.dtype
```

Out[20]:
```
dtype('float64')
```

In [21]:
```
#size can also be checked as.
a3d.size
```

Out[21]:
```
18
```

In [22]:
```
type(a3d)
```

Out[22]:
```
numpy.ndarray
```

In [23]:
```
import pandas as pd
```

In [24]:
```
df = pd.DataFrame(a2d)
```

In [25]:
```
df
```

Out[25]:

| | 0 | 1 | 2 |
|---|---|---|---|
| **0** | 1 | 2 | 3 |
| **1** | 1 | 2 | 3 |
| **2** | 1 | 2 | 3 |

# Creating Numpy Arrays

In [26]:
```
test_array = np.array([1,2,3])
```

In [27]:
```
test_array
```

Out[27]:
```
array([1, 2, 3])
```

automatically arrays can be created into three types in the form of either zeros, ones or random-numbers.

In [28]:
```
#np.ones((dimension/size))
ones=np.ones((2,3))
```

In [29]:
```
ones
```

Out[29]:
```
array([[1., 1., 1.],
       [1., 1., 1.]])
```

In [30]:
```
#result is in float.
ones.dtype
```

Out[30]:
```
dtype('float64')
```

In [31]:
```
type(ones)
```

Out[31]:
```
numpy.ndarray
```

In [32]:
```
zeros=np.zeros((4,3))
```

In [33]:
```
zeros
```

Out[33]:
```
array([[0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.]])
```

In [34]:
```
zeros.dtype
```

Out[34]:
```
dtype('float64')
```

An arange Concept

In [35]:
```
#Its like concept of range. such as arange(start, stop,step size)
range_array = np.arange(0,10,1)
```

In [36]:
```
range_array
```

Out[36]:
```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

# Creating Random Matrix

In [37]:
```
#np.random.randint(start, stop,size): randint:random integer. random is library
random_matrix = np.random.randint(0,10,size=(3,5))
```

In [38]:
```
random_matrix
```

Out[38]:
```
array([[5, 9, 4, 5, 0],
       [9, 2, 4, 9, 5],
       [5, 7, 6, 6, 7]])
```

In [39]:
```
#np.random.rand(number-of-rows, number-of-column). Rand remmains between zero to
rand_matrix = np.random.rand(2,5)
```

In [40]:
```
rand_matrix
```

Out[40]:
```
array([[0.25108145, 0.66933073, 0.61557277, 0.00153809, 0.24090699],
       [0.22459915, 0.40317043, 0.3348418 , 0.42839449, 0.5285026 ]])
```

In [41]:
```
rand_matrix.sum()
```

Out[41]:
```
3.697938498705465
```

In [42]:
```
#random, normally generates numbers of diff value. In order to fix it .seed() fur
np.random.seed(seed=0)
rand_matrix = np.random.rand(2,5)
```

In [43]:
```
rand_matrix.sum()
```

Out[43]:
```
6.157662833145425
```

# Accessing Elements of Arrays

In [44]:
```
a2 = np.array([[1,2,3],[3,4,5],[6,7,8]])
```

In [45]:
```
a2
```

Out[45]:
```
array([[1, 2, 3],
       [3, 4, 5],
       [6, 7, 8]])
```

In [46]:
```
# to access first row of 2D matrix.
a2[0]
```

Out[46]:
```
array([1, 2, 3])
```

In [47]:
```
# to access second row of 2D matrix
a2[1]
```

Out[47]:
```
array([3, 4, 5])
```

In [48]:
```
a3 = np.array([[[1,2,3],[4,5,6],[7,8,9]],[[1,2,3],[1,2,3],[1,2,3]]])
```

In [49]:
```
a3
```

Out[49]:
```
array([[[1, 2, 3],
        [4, 5, 6],
        [7, 8, 9]],

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

In [50]:
```
#to access first matrix of 3D matrix
a3[0]
```

Out[50]:
```
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

In [51]:
```
#to access 2nd matrix of 3D matrix
a3[1]
```

Out[51]:
```
array([[1, 2, 3],
       [1, 2, 3],
       [1, 2, 3]])
```

# Array Slicing

In [52]:
```
a3
```

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

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

In [53]:
```
# name-of-array[start-of-matrix:End-of-matrix,start-of-row:end-of-row,start-of-co
a3[0:2,0:2,0:2] # matrix started from zero end to 2, first two row of each matrix
```

Out[53]:
```
array([[[1, 2],
        [4, 5]],

       [[1, 2],
        [1, 2]]])
```

# Creating Complex Matrixes

In [54]:
```
arandom = np.random.randint(11,size=(2,3,4,5)) # size(replica of total number of
```

In [55]:
```
arandom
```

Out[55]:
```
array([[[[ 6,  7,  7,  8,  1],
         [ 5,  9,  8,  9,  4],
         [ 3,  0,  3,  5,  0],
         [ 2,  3,  8,  1,  3]],

        [[ 3,  3,  7,  0,  1],
         [ 9,  9,  0, 10,  4],
         [ 7,  3,  2,  7,  2],
         [ 0,  0,  4,  5,  5]],

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


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

        [[ 5,  9,  3,  0,  5],
         [ 0,  1,  2,  4,  2],
         [ 0,  3,  2, 10,  0],
         [ 7,  5,  9,  0, 10]],

        [[ 2, 10,  7,  2,  9],
         [ 2,  3,  3,  2,  3],
         [ 4,  1,  2,  9, 10],
         [ 1,  4, 10,  6,  8]]]])
```

In [56]:
```
#Accessing the first row of first matrix
arandom[0:1,0:1,0:1]
```

Out[56]:
```
array([[[[6, 7, 7, 8, 1]]]])
```

In [57]:
```
#Accessing the first element at first row and first matrix
arandom[0:1,0:1,0:1,0:1]
```

Out[57]:
```
array([[[[6]]]])
```

In [58]:
```
ones
```

Out[58]:
```
array([[1., 1., 1.],
       [1., 1., 1.]])
```

# Array Manipulation

In numpy, for loop is not applied but broadcasting/maping is carried out.

The algorithm which will be applied between the elements of two matrix is called broadcasting.

In [59]:
```
a1d
```

Out[59]:
```
array([1, 2, 3])
```

In [60]:
```
ones
```

Out[60]:
```
array([[1., 1., 1.],
       [1., 1., 1.]])
```

In [61]:
```
a1d + ones
```

Out[61]:
```
array([[2., 3., 4.],
       [2., 3., 4.]])
```

In [62]:
```
a1d - ones
```

Out[62]:
```
array([[0., 1., 2.],
       [0., 1., 2.]])
```

In [63]:
```
a1d*ones
```

Out[63]:
```
array([[1., 2., 3.],
       [1., 2., 3.]])
```

How to perform opperation upon incompatible matrix by creating some changes.

In [64]:
```
na3 = np.array([[[1,2,3,4],[4,5,6,5]],[[1,2,3,4],[1,2,3,5]]])
```

In [65]:
```
na3
```

Out[65]:
```
array([[[1, 2, 3, 4],
        [4, 5, 6, 5]],

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

In [66]:
```
a1d
```

Out[66]:
```
array([1, 2, 3])
```

In [67]:
```
a1d*na3
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-67-e2206d9ab50a> in <module>
----> 1 a1d*na3

ValueError: operands could not be broadcast together with shapes (3,) (2,2,4)
```

Error occurred bec of incompatibility between these two matrixes. So, we bring a few changes first in na3.

In [ ]:
```
t=na3[0:2,0:2,0:3]
```

In [ ]:
```
t
```

In [ ]:
```
t*a1d
```

In [ ]:
```
a1d/2
```

In [ ]:
```
a1d//2
```

In [ ]:
```
a1d**2
```

In [ ]:
```
np.square(a1d)
```

In [ ]:
```
a1d+ones
```

In [ ]:
```
np.add(a1d,ones)
```

In [ ]:
```
a1d%2
```

# Aggregation

Reducing the large set of data into single value is called aggregation. The function which performs the function of aggregation is called aggregate function.

In [ ]:
```
a1d
```

In [ ]:
```
sum(a1d) # here we didn't use the numpy library. So, this aggregate function is p
```

In [ ]:
```
np.sum(a1d) #Numpy method, faster method
```

if both are giving the same results, which method should be used? Let's check it.

In [68]:
```
n_array = np.random.random(1000)
```

In [69]:
```
n_array.size
```

Out[69]:
```
1000
```

In [70]:
```
n_array[:20] # taking only first 20 elements
```

Out[70]:
```
array([0.66741038, 0.13179786, 0.7163272 , 0.28940609, 0.18319136,
       0.58651293, 0.02010755, 0.82894003, 0.00469548, 0.67781654,
       0.27000797, 0.73519402, 0.96218855, 0.24875314, 0.57615733,
       0.59204193, 0.57225191, 0.22308163, 0.95274901, 0.44712538])
```

In [71]:
```
%timeit sum(n_array) #python_sum
%timeit np.sum(n_array) #numpy_sum
```

```
316 µs ± 20.1 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
14.2 µs ± 500 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

As depicted, python_sum is carried out in 1.22 miliseconds while numpy_sum is executed in 26.4 micro seconds. Therefore, numpy is far better.

In [72]:
```
np.max(a1d)
```

Out[72]:
```
3
```

In [73]:
```
np.min(a1d)
```

Out[73]:
```
1
```

# Mean, Variance and Standard Deviation.

Mean is considered to be the mi- point. Variance is the average difference of all values with mean. V = (x-x')/N , where x' is mean value, x is average difference from mean and N is total number of values. Moreover, standard deviation is the under-root of variance. V = ((x-x')/N )^1/2.

In [74]:
```
np.mean(a1d) #Mean value
```

Out[74]:
```
2.0
```

In [75]:
```
np.var(a1d) # variance
```

Out[75]:
```
0.6666666666666666
```

This means, collectively or on average, o.66 times all values are away from mean vlaue

In [76]:
```
np.sqrt(np.var(a1d)) #standard deviation
```

Out[76]:
```
0.816496580927726
```

In [77]:
```
#or
np.std(a1d)
```

Out[77]:
```
0.816496580927726
```

Use Shift+tab after higlighting to get help.

# Matrix Multiplication

Element wise multiplicaion: In this each element multiplies with the correspondent element of other matrix

In [78]:
```
np.random.seed(seed=0)
matrix_1 = np.random.randint(10,size=(4,3))
matrix_2 = np.random.randint(10,size=(4,3))
```

In [79]:
```
matrix_1
```

Out[79]:
```
array([[5, 0, 3],
       [3, 7, 9],
       [3, 5, 2],
       [4, 7, 6]])
```

In [80]:
```
matrix_2
```

Out[80]:
```
array([[8, 8, 1],
       [6, 7, 7],
       [8, 1, 5],
       [9, 8, 9]])
```

In [81]:
```
matrix_1*matrix_2
```

Out[81]:
```
array([[40,  0,  3],
       [18, 49, 63],
       [24,  5, 10],
       [36, 56, 54]])
```

Dot Product. Condition: no of columns of first must be equal to no of rows of second, and resultant will be equal to rows of first and columns of second matrix.

In [82]:
```
matrix_1.dot(matrix_2)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-82-70fe16dfc1dd> in <module>
----> 1 matrix_1.dot(matrix_2)

ValueError: shapes (4,3) and (4,3) not aligned: 3 (dim 1) != 4 (dim 0)
```

In [ ]:
```
t= matrix_2[0:3,0:3]
```

In [ ]:
```
t
```

In [ ]:
```
matrix_1.dot(t)
```

Transpose: It coverts the rows into column and column into rows.

In [ ]:
```
matrix_1.shape, matrix_2.T.shape
```

In [ ]:
```
#it can also be solved by converting the rows of any of above-stated two matrices
matrix_1.dot(matrix_2.T)
```

Transpose and Reshape:

In [ ]:
```
nm = np.array([[1,2,3],[5,7,8]])
```

In [ ]:
```
nm
```

In [ ]:
```
nm.T #TAKING TRANSPOSE
```

In [ ]:
```
#RESHAPING IT
nm.reshape(2,3,1)
```

# Exercize:

create random matrix with multiple number of sales of each item on each day and find the total sale of weak through .dot product.

In [ ]:
```
np.random.seed(seed=0)
item_amount = np.random.randint(20, size=(5,3))
```

In [ ]:
```
item_amount
```

In [ ]:
```
item_price = np.array([12,5,3])
```

In [ ]:
```
item_price
```

Uni-dimentional array doesnot exist. For that matter, array must have to be created by reshaping function with similar dimentions i.e(1,3)

In [ ]:
```
import pandas as pd
item_price_df = pd.DataFrame(item_price.reshape(1,3), index= ['item_price'], colu
item_price_df
```

In [ ]:
```
 item_amount_df = pd.DataFrame(item_amount, index=['M','T', 'W','TH','F'], columr
item_amount_df
```

In [ ]:
```
item_price_df.shape, item_amount_df.T.shape,
```

In [ ]:
```
per_day_sale= item_price_df.dot(item_amount_df.T)
per_day_sale
```

In [ ]:
```
item_amount_df['per_day_sale']= per_day_sale.T
item_amount_df
```

In [ ]:
```
item_price_df
```

# Comparison Operator

In [ ]:
```
a1d
```

In [ ]:
```
a2d
```

In [ ]:
```
a1d>a2d
```

In [ ]:
```
item_price
```

In [ ]:
```
a1d
```

In [ ]:
```
a1d<item_price
```

# Sort Method

In [86]:
```
np.random.seed(seed=0)
random_matrix = np.random.randint(10, size=(3,3))
random_matrix
```

Out[86]:
```
array([[5, 0, 3],
       [3, 7, 9],
       [3, 5, 2]])
```

In [87]:
```
np.sort(random_matrix) # generally it will sort rows in ascending order
```

Out[87]:
```
array([[0, 3, 5],
       [3, 7, 9],
       [2, 3, 5]])
```

In [89]:
```
np.sort(random_matrix, axis=1) #axis=1, another method to sort row, by default ax
```

Out[89]:
```
array([[0, 3, 5],
       [3, 7, 9],
       [2, 3, 5]])
```

In [91]:
```
np.sort(random_matrix, axis=0) #axis=0,  to sort column
```

Out[91]:
```
array([[3, 0, 2],
       [3, 5, 3],
       [5, 7, 9]])
```

ardsort(): it tells the indexes that how to organise/rearrange the data, or at particular place which value of which index should be place.

In [94]:
```
np.argsort(random_matrix, axis=1)
```

Out[94]:
```
array([[1, 2, 0],
       [0, 1, 2],
       [2, 0, 1]], dtype=int64)
```

In [95]:
```
np.argsort(random_matrix, axis=0)
```

Out[95]:
```
array([[1, 0, 2],
       [2, 2, 0],
       [0, 1, 1]], dtype=int64)
```

argmin: to know minimum value. argmax: to know max value

In [97]:
```
np.argmin(random_matrix) # will tell where minimum value is placed.
```

Out[97]:
```
1
```

In [98]:
```
w = np.array([[1,2,3],[4,5,6],[7,8,9]])# will tell where minimum value of matrix
```

In [99]:
```
w
```

Out[99]:
```
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

In [100]:
```
np.argmin(w)
```

Out[100]:
```
0
```

In [105]:
```
np.argmin(w, axis=1)# tells index value of each row where min value will be place
```

Out[105]:
```
array([0, 0, 0], dtype=int64)
```

In [112]:
```
np.argmax(w,axis=0)
```

Out[112]:
```
array([2, 2, 2], dtype=int64)
```

# Image Data Read

In [113]:
```
from matplotlib.image import imread
```

In [115]:
```
img_data = imread('11.jpg') # showing pixel value of an image.
```

In [116]:
```
img_data
```

Out[116]:
```
array([[[ 41,  55,  82],
        [ 41,  55,  82],
        [ 40,  54,  81],
        ...,
        [ 36,  17,   3],
        [ 33,  14,   0],
        [ 30,  10,   0]],

       [[ 38,  52,  79],
        [ 38,  52,  79],
        [ 38,  52,  79],
        ...,
        [ 36,  17,   3],
        [ 33,  14,   0],
        [ 30,  10,   0]],

       [[ 35,  49,  76],
        [ 36,  50,  77],
        [ 36,  50,  77],
        ...,
        [ 37,  18,   4],
        [ 33,  14,   0],
        [ 30,  10,   0]],

       ...,

       [[ 46,  35,  75],
        [ 42,  31,  71],
        [ 41,  30,  70],
        ...,
        [119, 105,  79],
        [114, 100,  74],
        [110,  96,  70]],

       [[ 39,  31,  68],
        [ 37,  29,  66],
        [ 40,  32,  69],
        ...,
        [121, 107,  81],
        [114, 100,  74],
        [109,  95,  69]],

       [[ 34,  28,  62],
        [ 34,  28,  62],
        [ 40,  34,  68],
        ...,
        [123, 109,  83],
        [115, 101,  75],
        [110,  96,  70]]], dtype=uint8)
```

In [119]:
```
#now to plot that image.
import matplotlib.pyplot as plt
```

In [122]:
```
plt.figure()
plt.imshow(img_data)
plt.show
```

Out[122]:
```
<function matplotlib.pyplot.show(close=None, block=None)>
```


In [123]:
```
img_data.shape #showing that it is three diesional where 3rd dimension is color.
```

Out[123]:
```
(960, 720, 3)
```

In [ ]:
```

```