numpy

Part-1

previously there was matlab(library/tool) which had multiple mathematical functions and python didn't had any so developer made **numpy** library (numerical python) inside python to perform all matrix and mathematical operation using python

In [47]:

```
import numpy as np
# we made np as alias for numpy, to shorten it for multiple use
```

matrix or array manipulation

In [48]:

```
# creating list
l = [1,2,3,4,5]
```

In [49]:

```
np.array(l)
#converted list into array,
# python list doesn't perform mathematical function, so need it in a array form
#that's why converted list into numpy array
```

Out[49]:

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

we can even convert array into list

In [50]:

```
arr = np.array(l)
```

In [51]:

```
type(arr) #nd array is n dimensional array
#numpy n dimensional array
```

Out[51]:

```
numpy.ndarray
```

In [52]:

```
np.asarray(l)
```

Out[52]:

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

- numpy.array()
- numpy.asarray()

both function will create numpy array

In [53]:

```
# passing 2 list in a numpy array
np.array([[1,2,3],[3,4,5]])
```

Out[53]:

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

it created 2 dimensional array

2 square bracket means 2 dimensional array which is 2 dimensional in nature

In [54]:
```
arr1 = np.array([[1,2,3],[2,3,4]])
```

In [55]:
```
type(arr1)
```

Out[55]:
```
numpy.ndarray
```

checking dimension of both single list and double list arrays

In [56]:
```
arr.ndim # 1 because it's 1 dimensional array
```

Out[56]:
```
1
```

In [57]:
```
arr1.ndim # 2 because it's a 2 dimensional array
```

Out[57]:
```
2
```

In [58]:
```
arr2 = np.array([[1,2,3],[2,3,4],[3,4,5]])
arr2
```

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

In [59]:
```
arr2.ndim
```

Out[59]:
```
2
```

this is how we plot any 2D array (numpy 2D array set)

numpy.matrix()

by default matrix create minimum of 2 dimensional array

matrix is sub-class of array

matrix is already a type of array

In [60]:
```
np.matrix(l)
```

Out[60]:
```
matrix([[1, 2, 3, 4, 5]])
```

In [61]:
```
mat = np.matrix(l)
```

numpy.asanyarray()

will create an array which is not in form of array it won't work/excute on any array

if we try to use **numpy.asanyarray()** for any matrix it won't do any thing because

- matrix is sub-call of array
- it's already a type of array

In [62]:
```
np.asanyarray(l) #create array because list was passed
```

Out[62]:
```
array([1, 2, 3, 4, 5])
```

In [63]:
```
np.asanyarray(mat) #didn't do anything because matrix was passed
```

Out[63]:
```
matrix([[1, 2, 3, 4, 5]])
```

to create array with numpy

- numpy.array()
- numpy.asarray()
- numpy.asanyarray()

any of these function will create n dimensional array

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

Out[64]:
```
array([1, 2, 3, 4, 5])
```

Shallow Copy

This happens because when you assign one variable to another in NumPy , you are creating a Shallow Copy. In other words, it is a reference to the same memory location. Therefore, by changing one variable the other one will change because they refer to the same memory location.

In [65]:
```
a = arr # assigning 'arr' array to 'a' variable
```

In [66]:
```
a
```

Out[66]:
```
array([1, 2, 3, 4, 5])
```

In [67]:
```
arr
```

Out[67]:
```
array([1, 2, 3, 4, 5])
```

In [68]:
```
#reassignment operation  on 'arr' array '0' position changing value '1' to 100
```

```
arr[0] = 100
```

```
In [69]: arr
```

```
Out[69]: array([100,   2,   3,   4,   5])
```

variable a[0] value has changed because we assigned arr array values refference to variable 'a'

```
In [70]: a
```

```
Out[70]: array([100,   2,   3,   4,   5])
```

```
In [71]: a[0] = 101
```

```
In [72]: a
```

```
Out[72]: array([101,   2,   3,   4,   5])
```

```
In [73]: arr
```

```
Out[73]: array([101,   2,   3,   4,   5])
```

again when we changed variable a[0] value to '101' it's got updated also for variable 'arr'

because we assigned only memory reference of array to both of the variable

- so whenerver one gets updated another will also reflect same updated values
- because both variable is pointing to same memory location or same address where array is stored

Deep copy

- numpy.copy() won't store memory reference of data/value,

This copy is completely a new array and copy owns the data. When we make changes to the copy it does not affect the original array, and when changes are made to the original array it does not affect the copy.

```
In [74]: b = np.copy(arr)
```

```
In [75]: b
```

```
Out[75]: array([101,   2,   3,   4,   5])
```

```
In [76]: b[0] = 234
```

```
In [77]: b
```

In [79]:

```
arr #it didn't change
```

Out[79]:

```
array([101,   2,   3,   4,   5])
```

create different - different kind of array

there are multiple functions to create diffn-diffn kind of array

In [ ]:

```
np.fromfunction()
#it takes function as an argument and based on that function or nature of function it's g
```

In [83]:

```
np.fromfunction( lambda i,j : i==j, (3,3)) #a data we are generating from given function
# here (3,3) is shape of array 3x3 : 3rows and 3 columns
```

Out[83]:

```
array([[ True, False, False],
       [False,  True, False],
       [False, False,  True]])
```

using fromfunction(), by writting own condition in it we can generated new kind of arrays

- ## creating array from function

In [85]:

```
np.fromfunction(lambda i,j : i*j, (3,3))
#it'll multiply row index with column index
```

Out[85]:

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

In [89]:

```
# using list comprehension operation
(i*i for i in range(5))
```

Out[89]:

```
<generator object <genexpr> at 0x7f400f6bcf90>
```

In [91]:

```
list(i*i for i in range(5))
```

now if we want to execute same kind of operation with numpy

In [93]:

```
iterable = (i*i for i in range(5)) #enclosed inside tuple
```

In [94]:

```
np.fromiter(iterable)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[94], line 1
----> 1 np.fromiter(iterable)

TypeError: fromiter() missing required argument 'dtype' (pos 2)
```

- ## creating array from iterable

In [95]:

```
np.fromiter(iterable, float) #have to pass datatype also
```

Out[95]:

```
array([ 0.,  1.,  4.,  9., 16.])
```

- ## creating array from string

In [97]:

```
np.fromstring('23 45 56',sep = ' ') #here seperator is space ' '
```

Out[97]:

```
array([23., 45., 56.])
```

In [99]:

```
np.fromstring('23,45,56', sep = ',') # seperator is comma ','
```

Out[99]:

```
array([23., 45., 56.])
```

In [100…

```
arr
```

Out[100…

```
array([101,   2,   3,   4,   5])
```

In [101…

```
arr1
```

Out[101…

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

checking dimension of an array

In [104…

```
arr.ndim
```

Out[104…

```
1
```

In [105…

```
arr1.ndim
```

Out[105…

```
2
```

checking size of array, size is total no. of element in array

In [106…

```
arr.size
```

Out[106…

```
5
```

In [107…

```
arr1.size
```

Out[107…

```
6
```

checking shape of array, no. of rows and columns

for single dimension it'll show no. of 'elements,' only

In [108…

```
arr.shape
```

Out[108…

```
(5,)
```

In [109…

```
arr1.shape
```

Out[109…

```
(2, 3)
```

In [110…

```
arr1
```

Out[110…

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

checking datatype of array

In [111…

```
arr.dtype
```

Out[111…

```
dtype('int64')
```

In [112…

```
arr1.dtype
```

Out[112…

```
dtype('int64')
```

In [ ]:

```

```

-

-

-

-

-

Part-2

In [113

```
import numpy as np
```

In [114…

```
range(5)
```

Out[114…

```
range(0, 5)
```

In [115…

```
# passing inside list() create list-data of number from 0 to 1
list(range(5))
```

Out[115…

```
[0, 1, 2, 3, 4]
```

In [116…

```
list(range(0,10))
```

Out[116…

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

numpy.arange()

for fractional value range() will not create list of fraction values

range() only take/consider integer value

In [117…

```
# for fraction
list(range(0.4,10.4))
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[117], line 2
      1 # for fraction
----> 2 list(range(0.4,10.4))

TypeError: 'float' object cannot be interpreted as an integer
```

numpy has arange() function to create range of numbers array, even fraction number

In [118…

```
np.arange(.4,10.4)
```

Out[118…

```
array([0.4, 1.4, 2.4, 3.4, 4.4, 5.4, 6.4, 7.4, 8.4, 9.4])
```

In [120…

```
np.arange(0.4, 10.4, 2) #2 is for steps
# it'll will skip 2 steps and print
```

Out[120…

```
array([0.4, 2.4, 4.4, 6.4, 8.4])
```

In [122…

```
np.arange(.4 , 10.4, 0.2)
# printing after 0.2 steps
```

Out[122…

```
array([ 0.4,  0.6,  0.8,  1. ,  1.2,  1.4,  1.6,  1.8,  2. ,  2.2,  2.4,
        2.6,  2.8,  3. ,  3.2,  3.4,  3.6,  3.8,  4. ,  4.2,  4.4,  4.6,
        4.8,  5. ,  5.2,  5.4,  5.6,  5.8,  6. ,  6.2,  6.4,  6.6,  6.8,
        7. ,  7.2,  7.4,  7.6,  7.8,  8. ,  8.2,  8.4,  8.6,  8.8,  9. ,
        9.2,  9.4,  9.6,  9.8, 10. , 10.2])
```

converting array to a list

In [123…

```
list(np.arange( .4, 10.4, 0.2))
```

Out[123…

```
[0.4,
 0.6000000000000001,
 0.8000000000000002,
 1.0000000000000002,
 1.2000000000000002,
 1.4000000000000004,
 1.6000000000000005,
 1.8000000000000003,
 2.0000000000000004,
 2.2000000000000006,
 2.400000000000001,
 2.6000000000000005,
 2.8000000000000007,
 3.000000000000001,
 3.2000000000000006,
 3.400000000000001,
 3.600000000000001,
 3.800000000000001,
 4.000000000000002,
 4.200000000000001,
 4.400000000000002,
 4.600000000000001,
 4.800000000000002,
 5.000000000000002,
 5.200000000000002,
 5.400000000000002,
 5.600000000000002,
 5.8000000000000025,
 6.000000000000002,
 6.200000000000002,
 6.400000000000002,
 6.600000000000002,
 6.8000000000000025,
 7.000000000000003,
 7.200000000000003,
 7.400000000000003,
 7.600000000000003,
 7.8000000000000025,
 8.000000000000002,
 8.200000000000003,
 8.400000000000004,
 8.600000000000003,
 8.800000000000002,
 9.000000000000004,
 9.200000000000003,
 9.400000000000004,
 9.600000000000003,
 9.800000000000004,
 10.000000000000004,
 10.200000000000003]
```

numpy.linspace()

- Return evenly spaced numbers over a specified interval.
- The numpy.linspace() function is used to create an array of evenly spaced numbers within a specified range. The range is defined by the start and end points of the sequence, and the number of evenly spaced points to be generated between them.

In [126…

```
#in a scale of 1 to 5,  create 20 such data
# it'll divide 1 to 5 in 20 division equally to generate 20 number of data
np.linspace(1,5, 20)
#it'll create array between 1 to 5, with equal step tobe of 20 numbers
```

Out[126…

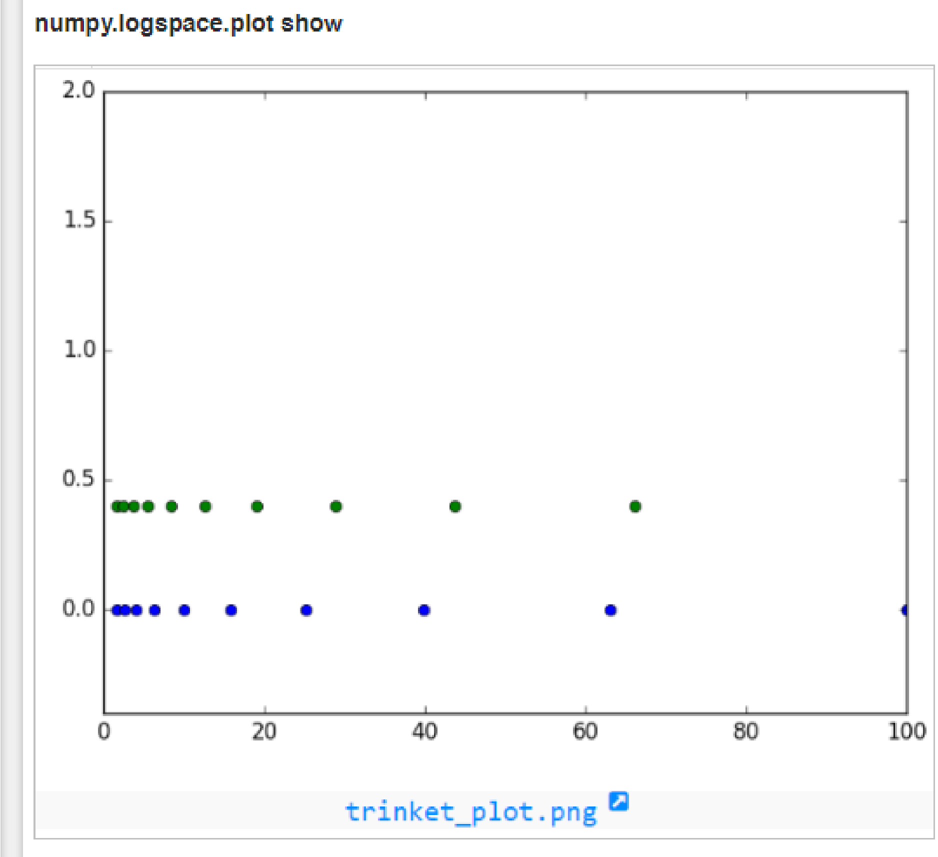
```
array([1.        , 1.21052632, 1.42105263, 1.63157895, 1.84210526,
       2.05263158, 2.26315789, 2.47368421, 2.68421053, 2.89473684,
       3.10526316, 3.31578947, 3.52631579, 3.73684211, 3.94736842,
       4.15789474, 4.36842105, 4.57894737, 4.78947368, 5.        ])
```

numpy.logspace()

The numpy.logspace() function returns an array with numbers that are evenly spaced on a logarithmic scale. It is similar to linspace() but instead of linearly spaced values, logspace() returns values spaced logarithmically.

In [127…
```
np.logspace(1,5, 10)
```

Out[127…
```
array([1.00000000e+01, 2.78255940e+01, 7.74263683e+01, 2.15443469e+02,
       5.99484250e+02, 1.66810054e+03, 4.64158883e+03, 1.29154967e+04,
       3.59381366e+04, 1.00000000e+05])
```



can define base also, bydefault it's 10

In [129…
```
np.logspace(1,5, 10, base=10)
```

Out[129…
```
array([1.00000000e+01, 2.78255940e+01, 7.74263683e+01, 2.15443469e+02,
       5.99484250e+02, 1.66810054e+03, 4.64158883e+03, 1.29154967e+04,
       3.59381366e+04, 1.00000000e+05])
```

In [130…
```
np.logspace(1,5, 10, base=2) # now base of log is 2
```

Out[130…
```
array([ 2.        ,  2.72158   ,  3.70349885,  5.0396842 ,  6.85795186,
        9.33223232, 12.69920842, 17.28095582, 23.51575188, 32.        ])
```

numpy.zeros()

generate array or matrix with zeros with n-rows and n-columns

In [138…
```
# it'll createllll 5 data with all zero
np.zeros(5)
```

Out[138…

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

In [139…

```
np.zeros((3,4)) #it's no. of rows and columns
#basically shape of zeros data
```

Out[139…

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

In [140…

```
np.zeros((3,4,2))
#it'll create 3Dimensional data of 3 same copy of 4rows and 2columns
```

Out[140…

```
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.]]])
```

np.zeros((3,4,2)), create/generate 3 matrix with 4rows and 2columns data

3 dimensional data has z axis, (kind a behind 2D plane)

numpy.zeros can generate any number of dimensional array but only till 3 dimension we can visualize it

In [141…

```
np.zeros((3,4,2,3))
```

Out[141…

```
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., 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.]]]])
```

numpy.ones()

it'll create ndimensional arrays or matrix with 1 values inside

In [145…

```
np.ones(5) #generate array of five element with 1,1,1,1,1 in it
```

Out[145…

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

In [147…

```
np.ones((3,4)) # 3rows and 4column
```

Out[147…

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

In [148…

```
arr= np.ones((3,4))
```

In [149…

```
arr
```

Out[149…

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

In [151…

```
#adding a array(matrix)
arr+5
# it'll add 5 in each element
```

Out[151…

```
array([[6., 6., 6., 6.],
       [6., 6., 6., 6.],
       [6., 6., 6., 6.]])
```

In [153…

```
arr*4
#multiply 4 with each element and return array
```

Out[153…

```
array([[4., 4., 4., 4.],
       [4., 4., 4., 4.],
       [4., 4., 4., 4.]])
```

numpy.empty()

In [154…

```
np.empty(3)
```

Out[154...

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

In [155...

```
np.empty((3,4))
```

Out[155...

```
array([[4., 4., 4., 4.],
       [4., 4., 4., 4.],
       [4., 4., 4., 4.]])
```

numpy.eye()

- it's an identity matrix, because it'll create matrix and '1' will be in diagonally
- and determinant of identity matrix is always 1

The determinant of the identity matrix is 1; the exchange of two rows (or of two columns) multiplies the determinant by −1; multiplying a row (or a column) by a number multiplies the determinant by this number; and adding to a row (or a column) a multiple of another row (or column) does not change the determinant.

In [158...

```
np.eye(5) # it maybe only takes single integer as argument
#no. of rows will always be equal to no. of columns
```

Out[158...

```
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.]])
```

In [160...

```
np.eye((3,4))# identity matrix always be a square matrix,
#no. of rows will always be equal to no. of columns
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[160], line 1
----> 1 np.eye((3,4))# identity matrix always be a square matrix,
      2 #no. of rows will always be equal to no. of columns

File /opt/conda/lib/python3.10/site-packages/numpy/lib/twodim_base.py:215, in eye(N, M, k,
dtype, order, like)
    213 if M is None:
    214     M = N
--> 215 m = zeros((N, M), dtype=dtype, order=order)
    216 if k >= M:
    217     return m

TypeError: 'tuple' object cannot be interpreted as an integer
```

In [163...

```
 
```

converting numpy created data to pandas DataFrame

In [165...

```
import pandas as pd

arr1 = np.eye(5)
pd.DataFrame(arr1)
```

Out[165...

	0	1	2	3	4
0	1.0	0.0	0.0	0.0	0.0

1	0.0	1.0	0.0	0.0	0.0
2	0.0	0.0	1.0	0.0	0.0
3	0.0	0.0	0.0	1.0	0.0
4	0.0	0.0	0.0	0.0	1.0

numpy.random.rand()

In [167…

```
np.random.rand(2,3) # it'll generate a data with randon number of 2rows and 3columns
#it'll generate data where mean(), standard deviation can be any thing
```

Out[167…

```
array([[0.26466688, 0.78166358, 0.26152883],
       [0.64904497, 0.44117883, 0.32094182]])
```

numpy.random.randn()

In [170…

```
np.random.randn(2,3) # it'll also generate a data with randon number of 2rows and 3colum
# it'll generate a data with standard normal distribution
# where means=0 and standard deviation=1

#will use it when perform some statistical operation
```

Out[170…

```
array([[ 0.94902541, -0.16057205, -1.89077257],
       [ 0.91514816, -0.43859375, -1.90192781]])
```

np.random.randint()

In [171…

```
np.random.randint(1,5, (3,4)) #range from 1to5, and shape is (3,4)-3rows and 4columns
```

Out[171…

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

will use for data manipulation and many statistical operation where we need random number/data of any size and any shape

In [172…

```
arr2 = np.random.randint(1,5, (3,4))
```

In [173…

```
arr2
```

Out[173…

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

In [174…

```
arr2.size
```

Out[174…

```
12
```

In [175…

```
arr2.shape
```

Out[175…

```
(3, 4)
```

numpy.reshape()

changing shape of array/matrix

In [177…

```
#when giving new shape, the no. of data/element should not be change
arr2.reshape(4,5)
#because 4x5 array/matrix has more than 12 elements so it won't work
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[177], line 2
      1 #when giving new shape, the no. of data/element should not be change
----> 2 arr2.reshape(4,5)
      3 #because 4x5 array/matrix has more than 12 elements so it won't work

ValueError: cannot reshape array of size 12 into shape (4,5)
```

In [178…

```
arr2.reshape(4,3)
```

Out[178…

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

In [179…

```
arr2
```

Out[179…

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

In [180…

```
arr2.reshape(2,6)
```

Out[180…

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

In [184…

```
arr2.reshape(6,2)
```

Out[184…

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

In [183…

```
arr2.reshape(2,4)#again no. of elements will not be same as is arr2
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[183], line 1
----> 1 arr2.reshape(2,4)#again no. of elements will not be same as is arr2

ValueError: cannot reshape array of size 12 into shape (2,4)
```

In [185…

```
arr2.reshape(4,-1)
#it understood 4rows so column calculate by it self
```

Out[185…

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

In [188…

```
arr2.reshape(4,-1324346378469836)
# can give any negative number
#but it'll reshape what ever column is needed to complete/reshape
#the array/matrix
```

Out[188…

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

can reshape to any number of dimension

In [191…

```
arr2.reshape(2,2,3)
#2x2x3 = 12
# created 3 dimensional array with 12 no. of element
```

Out[191…

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

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

In [193…

```
arr2.reshape(2,2,3,1,1,1)
# it can generate n number of n-dimensional array
```

Out[193…

```
array([[[[[[4]]],


         [[[1]]],


         [[[1]]]],



        [[[[3]]],


         [[[1]]],


         [[[2]]]]],




       [[[[[2]]],


         [[[1]]],


         [[[1]]]],



        [[[[3]]],


         [[[4]]],


         [[[4]]]]]])
```

In [194…

```
arr1 = np.random.randint(1,10, (5,6))
```

In [195…

```
arr1
```

Out[195…

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

In [198…

```
# filtering which element of array is greater than 8
arr1>8
# where element is greater than 8 it'll return True or otherwise False
```

Out[198…

```
array([[False,  True, False,  True, False, False],
       [False, False, False, False, False, False],
       [False, False, False, False,  True, False],
       [False, False, False, False, False, False],
       [False, False, False, False, False,  True]])
```

In [199…

```
#print only where arr1>8
arr1[arr1>8]
```

Out[199…

```
array([9, 9, 9, 9])
```

In [200…

```
arr1
```

Out[200…

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

In [202…

```
arr1[0,0:2] # only extracting value of 1st row's column1 and column2
# extracting subset of data by default index given by python
```

Out[202…

```
array([2, 9])
```

In [203…

```
arr1[0]
```

Out[203…

```
array([2, 9, 7, 9, 1, 2])
```

In [205…

```
arr1[0,[0,1]] #first row and inside it, first and second column
# or arr1[0,0:2]
```

Out[205…

```
array([2, 9])
```

In [208…

```
arr1[2:4 ,[2,3]]
```

Out[208…

```
array([[7, 8],
       [8, 4]])
```

- so it's same as list slicing and indexing

In [ ]:

```

```

In [211…

```
arr1 = np.random.randint(1,3, (3,3))
arr2 = np.random.randint(1,3, (3,3))
```

In [212…

```
arr1
```

Out[212…

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

In [213…

```
arr2
```

Out[213…

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

In [217…

```
# it'll perform index wise addition between both arrays/matrix
# it's not matrix wise operation
arr1 + arr2
```

Out[217…

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

In [215…

```
arr1 - arr2
```

Out[215…

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

In [216…

```
arr1*arr2
```

Out[216…

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

so add,sub, multiply, will perform index-wise addition....

it's not matrix-wise

matrix multiplication

- should be same rows and columns
- and perform between rows and columns

> multiply row 1st row with 1st column and add all to get a element

In [219…

```
arr1
```

Out[219…

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

In [220…

```
arr2
```

Out[220…

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

In [221…

```
arr1@arr2 # matrix multiplication
```

Out[221…

```
array([[5, 7, 5],
       [6, 8, 7],
       [4, 5, 4]])
```

In [ ]:

In [222…

```
arr1/arr2
```

Out[222…

```
array([[0.5, 0.5, 2. ],
       [1. , 2. , 1. ],
       [1. , 0.5, 1. ]])
```

In [224…

```
arr/0 #divide by 0 does exist inside numpy but not in python core
```

```
arr/0 #divide by 0 does exist inside numpy but not in python core
#because it's returning infinite:'inf'
```

```
/tmp/ipykernel_154/2405269321.py:1: RuntimeWarning: divide by zero encountered in divide
  arr/0 #divide by 0 does exist inside numpy but not in python core
```

Out[224…

```
array([[inf, inf, inf],
       [inf, inf, inf],
       [inf, inf, inf]])
```

Broadcasting operation

In [226…

```
arr = np.zeros((3,4))
```

In [227…

```
arr
```

Out[227…

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

In [230…

```
arr+5 #giving 1 value but it's using/putting in each element inside array
```

Out[230…

```
array([[5., 5., 5., 5.],
       [5., 5., 5., 5.],
       [5., 5., 5., 5.]])
```

adding (3,4)array with 1D array

- column wise addition

In [233…

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

In [234…

```
a
```

Out[234…

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

In [236…

```
arr+a #arr has zeros and it's adding column wise in array
```

Out[236…

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

- row-wise addition

In [244…

```
b = np.array([3,4,5])
```

In [245…

```
b
```

Out[245…

```
array([3, 4, 5])
```

In [246…

```
arr+b
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[246], line 1
----> 1 arr+b

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

In [247…

```
arr+b.T
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[247], line 1
----> 1 arr+b.T

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

In [248…

```
b.ndim
```

Out[248…

```
1
```

In [249…

```
b = np.array([[3,4,5]])
```

In [251…

```
b
```

Out[251…

```
array([[3, 4, 5]])
```

In [252…

```
b.ndim
```

Out[252…

```
2
```

In [255…

```
b.T #transposing b array column to row
```

Out[255…

```
array([[3],
       [4],
       [5]])
```

In [256…

```
arr+b.T #transposing b array column to row
```

Out[256…

```
array([[3., 3., 3., 3.],
       [4., 4., 4., 4.],
       [5., 5., 5., 5.]])
```

In [ ]:

In [257…

```
arr1 = arr+b.T
```

In [258…

```
arr1
```

Out[258…

```
array([[3., 3., 3., 3.],
       [4., 4., 4., 4.],
       [5., 5., 5., 5.]])
```

numpy sqrt() :square root of array

In [260…

```
np.sqrt(arr1) #for every element it'll do square_root
```

Out[260…

```
array([[1.73205081, 1.73205081, 1.73205081, 1.73205081],
       [2.        , 2.        , 2.        , 2.        ],
       [2.23606798, 2.23606798, 2.23606798, 2.23606798]])
```

numpy log10()

In [262…

```
np.log10(arr1) # return log of base 10 of every element
```

Out[262…

```
array([[0.47712125, 0.47712125, 0.47712125, 0.47712125],
```

Out[262…

```
array([[0.47712125, 0.47712125, 0.47712125, 0.47712125],
       [0.60205999, 0.60205999, 0.60205999, 0.60205999],
       [0.69897   , 0.69897   , 0.69897   , 0.69897   ]])
```

numpy exponent()

In [264…

```
np.exp(arr1)
```

Out[264…

```
array([[ 20.08553692,  20.08553692,  20.08553692,  20.08553692],
       [ 54.59815003,  54.59815003,  54.59815003,  54.59815003],
       [148.4131591 , 148.4131591 , 148.4131591 , 148.4131591 ]])
```

numpy min() minimum

In [265…

```
np.min(arr1)
```

Out[265…

```
3.0
```

numpy max() maximum

In [266…

```
np.max(arr1)
```

Out[266…

```
5.0
```

-

-

-

-

-

Part-3

In [2]:

```
import numpy as np
```

Numpy - Array Manipulation.

In [3]:

```
arr = np.random.randint(1,10, (4,4))
```

In [4]:

```
arr
```

Out[4]:

```
array([[9, 1, 9, 5],
       [6, 7, 6, 6],
       [2, 8, 7, 2],
       [5, 8, 9, 8]])
```

In [5]:

```
arr.reshape(8,2)
```

Out[5]:

```
array([[9, 1],
       [9, 5],
       [6, 7],
       [6, 6],
       [2, 8],
       [7, 2],
       [5, 8],
       [9, 8]])
```

In [6]:

```
arr
```

Out[6]:

```
array([[9, 1, 9, 5],
       [6, 7, 6, 6],
       [2, 8, 7, 2],
       [5, 8, 9, 8]])
```

In [7]:

```
arr.T #tranpose
```

Out[7]:

```
array([[9, 6, 2, 5],
       [1, 7, 8, 8],
       [9, 6, 7, 9],
       [5, 6, 2, 8]])
```

In [15]:

```
arr.flatten() # convert it into single list/1D array
```

Out[15]:

```
array([9, 1, 9, 5, 6, 7, 6, 6, 2, 8, 7, 2, 5, 8, 9, 8])
```

In [16]:

```
type(arr.flatten())
```

Out[16]:

```
numpy.ndarray
```

In [17]:

```
type(arr)
```

Out[17]:

```
numpy.ndarray
```

In [18]:

```
arr
```

Out[18]:

```
array([[9, 1, 9, 5],
       [6, 7, 6, 6],
       [2, 8, 7, 2],
       [5, 8, 9, 8]])
```

In [19]:

```
np.expand_dims(arr)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[19], line 1
----> 1 np.expand_dims(arr)

File <__array_function__ internals>:179, in expand_dims(*args, **kwargs)

TypeError: _expand_dims_dispatcher() missing 1 required positional argument: 'axis'
```

In [22]:

```
np.expand_dims(arr, axis=1) #expanded dimension from 2D to 3D
#from 2-dimension to  3-dimension
#across column it's expanded dimension, because axis=1
```

Out[22]:

```
array([[[9, 1, 9, 5]],

       [[6, 7, 6, 6]],
       [[2, 8, 7, 2]],

       [[5, 8, 9, 8]]]])
```

In [24]:
```
np.expand_dims(arr, axis=0)
# expanded dimension across rows, because axis=0
```

Out[24]:
```
array([[[[9, 1, 9, 5],
         [6, 7, 6, 6],
         [2, 8, 7, 2],
         [5, 8, 9, 8]]]])
```

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

In [4]:
```
data
```

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

In [6]:
```
np.squeeze(data) #from 2D to 1D
```

Out[6]:
```
array([1, 2, 3])
```

In [11]:
```
np.repeat(data,2) #repeat data how many times we want
```

Out[11]:
```
array([1, 1, 2, 2, 3, 3])
```

In [12]:
```
np.repeat(data,3)
```

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

In [15]:
```
np.roll(data,1)
```

Out[15]:
```
array([[3],
       [1],
       [2]])
```

In [17]:
```
np.roll(data,2)
#roll data to 2 step further, now 1 is 2 step further from pervious position
```

Out[17]:
```
array([[2],
       [3],
       [1]])
```

In [20]:
```
np.diag(np.array([1,2,3,4]) ) # in a 2D square matrix it'll place data diagonally
```

Out[20]:
```
array([[1, 0, 0, 0],
       [0, 2, 0, 0],
       [0, 0, 3, 0],
       [0, 0, 0, 4]])
```

numpy- Binary Operations.

- addition, substraction, multiplication

In [23]:
```
arr1 = np.random.randint(1,10, (3,4))
arr2 = np.random.randint(1,10, (3,4))
```

```
In [24]: arr1
```

```
Out[24]: array([[8, 8, 5, 7],
                [1, 4, 4, 6],
                [9, 7, 7, 6]])
```

```
In [25]: arr2
```

```
Out[25]: array([[8, 1, 8, 3],
                [1, 8, 6, 7],
                [9, 1, 6, 1]])
```

index-wise operations

```
In [27]: arr1+arr2 #index-wise summation operation
```

```
Out[27]: array([[16,  9, 13, 10],
                [ 2, 12, 10, 13],
                [18,  8, 13,  7]])
```

```
In [28]: arr1*arr2
```

```
Out[28]: array([[64,  8, 40, 21],
                [ 1, 32, 24, 42],
                [81,  7, 42,  6]])
```

```
In [29]: arr1/arr2
```

```
Out[29]: array([[1.        , 8.        , 0.625     , 2.33333333],
                [1.        , 0.5       , 0.66666667, 0.85714286],
                [1.        , 7.        , 1.16666667, 6.        ]])
```

```
In [30]: arr1-arr2
```

```
Out[30]: array([[ 0,  7, -3,  4],
                [ 0, -4, -2, -1],
                [ 0,  6,  1,  5]])
```

```
In [31]: arr1**arr2
```

```
Out[31]: array([[ 16777216,         8,    390625,       343],
                [        1,     65536,      4096,    279936],
                [387420489,         7,    117649,         6]])
```

negation of array(convert into negative numbers)

```
In [34]: ~arr1 #binary negation operation with '~'
```

```
Out[34]: array([[ -9,  -9,  -6,  -8],
                [ -2,  -5,  -5,  -7],
                [-10,  -8,  -8,  -7]])
```

```
In [35]: arr1
```

```
Out[35]: array([[8, 8, 5, 7],
                [1, 4, 4, 6],
                [9, 7, 7, 6]])
```

```
In [36]: arr1>arr2
```

```
Out[36]: array([[False,  True, False,  True],
                [False, False, False, False],
```

numpy- String Operations

it has all string operations that we use in python

> upper, lower, capitalize, title

In [37]:

```
arr = np.array(["resheph", "RR"])
```

In [38]:

```
arr
```

Out[38]:

```
array(['resheph', 'RR'], dtype='<U7')
```

turning numpy array string into upper character

In [39]:

```
np.char.upper(arr)
```

Out[39]:

```
array(['RESHEPH', 'RR'], dtype='<U7')
```

In [40]:

```
np.char.capitalize(arr)
```

Out[40]:

```
array(['Resheph', 'Rr'], dtype='<U7')
```

In [42]:

```
np.char.title(arr)
```

Out[42]:

```
array(['Resheph', 'Rr'], dtype='<U7')
```

numpy- Mathematical Functions.

In [43]:

```
arr1
```

Out[43]:

```
array([[8, 8, 5, 7],
       [1, 4, 4, 6],
       [9, 7, 7, 6]])
```

In [47]:

```
np.sin(arr1) #find sin
```

Out[47]:

```
array([[ 0.98935825,  0.98935825, -0.95892427,  0.6569866 ],
       [ 0.84147098, -0.7568025 , -0.7568025 , -0.2794155 ],
       [ 0.41211849,  0.6569866 ,  0.6569866 , -0.2794155 ]])
```

In [45]:

```
np.cos(arr1)
```

Out[45]:

```
array([[-0.14550003, -0.14550003,  0.28366219,  0.75390225],
       [ 0.54030231, -0.65364362, -0.65364362,  0.96017029],
       [-0.91113026,  0.75390225,  0.75390225,  0.96017029]])
```

In [46]:

```
np.tan(arr1)
```

Out[46]:

```
array([[-6.79971146, -6.79971146, -3.38051501,  0.87144798],
       [ 1.55740772,  1.15782128,  1.15782128, -0.29100619],
       [-0.45231566,  0.87144798,  0.87144798, -0.29100619]])
```

In [48]:

```
np.log10(arr1)
```

```
Out[48]: array([[0.90308999, 0.90308999, 0.69897   , 0.84509804],
                [0.        , 0.60205999, 0.60205999, 0.77815125],
                [0.95424251, 0.84509804, 0.84509804, 0.77815125]])
```

```
In [49]: np.log2(arr1)
```

```
Out[49]: array([[3.        , 3.        , 2.32192809, 2.80735492],
                [0.        , 2.        , 2.        , 2.5849625 ],
                [3.169925  , 2.80735492, 2.80735492, 2.5849625 ]])
```

```
In [52]: np.exp(arr1) # find exponent of data
```

```
Out[52]: array([[2.98095799e+03, 2.98095799e+03, 1.48413159e+02, 1.09663316e+03],
                [2.71828183e+00, 5.45981500e+01, 5.45981500e+01, 4.03428793e+02],
                [8.10308393e+03, 1.09663316e+03, 1.09663316e+03, 4.03428793e+02]])
```

```
In [55]: np.power(arr1,2) # find power of 2, we can put any interger to find power
```

```
Out[55]: array([[64, 64, 25, 49],
                [ 1, 16, 16, 36],
                [81, 49, 49, 36]])
```

```
In [57]: np.mean(arr1) #calculate average of whole array
```

```
Out[57]: 6.0
```

```
In [59]: np.median(arr1) # find middle value of entire array
```

```
Out[59]: 6.5
```

```
In [63]: np.mode(arr1) #numpy doesn't has mode() function
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Cell In[63], line 1
----> 1 np.mode(arr1) #numpy doesn't has mode() function

File /opt/conda/lib/python3.10/site-packages/numpy/__init__.py:311, in __getattr__(attr)
    308     from .testing import Tester
    309     return Tester
--> 311 raise AttributeError("module {!r} has no attribute "
    312                      "{!r}".format(__name__, attr))

AttributeError: module 'numpy' has no attribute 'mode'
```

```
In [65]: np.std(arr1) # find standard deviation
```

```
Out[65]: 2.1213203435596424
```

```
In [67]: np.var(arr1) # find variance
```

```
Out[67]: 4.5
```

```
In [68]: np.min(arr1)
```

```
Out[68]: 1
```

```
In [69]: np.max(arr1)
```

```
Out[69]: 9
```

numpy- Arithmetic Operations.

> subtract, addition, modulus..

In [71]:
```
arr1
```

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

In [72]:
```
arr2
```

Out[72]:
```
array([[8, 1, 8, 3],
       [1, 8, 6, 7],
       [9, 1, 6, 1]])
```

In [75]:
```
arr1-arr2
```

Out[75]:
```
array([[ 0,  7, -3,  4],
       [ 0, -4, -2, -1],
       [ 0,  6,  1,  5]])
```

In [76]:
```
np.subtract(arr1, arr2) # both are same either '-' or 'numpy.substract()'
```

Out[76]:
```
array([[ 0,  7, -3,  4],
       [ 0, -4, -2, -1],
       [ 0,  6,  1,  5]])
```

In [77]:
```
arr1*arr2
```

Out[77]:
```
array([[64,  8, 40, 21],
       [ 1, 32, 24, 42],
       [81,  7, 42,  6]])
```

In [78]:
```
np.multiply(arr1,arr2)
```

Out[78]:
```
array([[64,  8, 40, 21],
       [ 1, 32, 24, 42],
       [81,  7, 42,  6]])
```

In [81]:
```
arr1%arr2 #return remainder after divide
```

Out[81]:
```
array([[0, 0, 5, 1],
       [0, 4, 4, 6],
       [0, 0, 1, 0]])
```

In [82]:
```
np.mod(arr1,arr2)
```

Out[82]:
```
array([[0, 0, 5, 1],
       [0, 4, 4, 6],
       [0, 0, 1, 0]])
```

In [84]:
```
arr1**arr2
```

Out[84]:
```
array([[ 16777216,         8,    390625,       343],
       [        1,     65536,      4096,    279936],
       [387420489,         7,    117649,         6]])
```

In [83]:
```
np.power(arr1,arr2)
```

Out[83]:
```
array([[ 16777216,         8,    390625,       343],
       [        1,     65536,      4096,    279936],
       [387420489,         7,    117649,         6]])
```

In [86]:

```
np.sqrt(arr1) #square root
```

Out[86]:

```
array([[2.82842712, 2.82842712, 2.23606798, 2.64575131],
       [1.        , 2.        , 2.        , 2.44948974],
       [3.        , 2.64575131, 2.64575131, 2.44948974]])
```

numpy- Statistical Functions.

> mean, median, mode

In [87]:

```
arr1
```

Out[87]:

```
array([[8, 8, 5, 7],
       [1, 4, 4, 6],
       [9, 7, 7, 6]])
```

In [88]:

```
np.mean(arr1)
```

Out[88]:

```
6.0
```

In [89]:

```
np.std(arr1) #dispersion from the mean
```

Out[89]:

```
2.1213203435596424
```

In [90]:

```
np.median(arr1)
```

Out[90]:

```
6.5
```

-

-

-

-

-

Part-4

In [91]:

```
import numpy as np
```

Sort, Search & Counting Functions.

In [100…

```
arr = np.array([4,2,8,5,3,9,12,56])
```

```
In [101…  arr

Out[101…  array([ 4,  2,  8,  5,  3,  9, 12, 56])
```

```
In [102…  print(arr.sort())

None
```

```
In [104…  np.sort(arr) # sort in ascending order

Out[104…  array([ 2,  3,  4,  5,  8,  9, 12, 56])
```

```
In [105…  np.sort

Out[105…  <function numpy.sort(a, axis=-1, kind=None, order=None)>
```

```
In [106…  np.searchsorted(arr,6) # it'll search index where we can put given data in
          # in which place of array it can put data to maintain

Out[106…  4
```

```
In [108…  arr1 = np.array([0,324,645,65,6,6,0,0,0,234])
```

```
In [109…  arr1

Out[109…  array([  0, 324, 645,  65,   6,   6,   0,   0,   0, 234])
```

```
In [111…  np.count_nonzero(arr1) # how many element doesn't have zero

Out[111…  6
```

```
In [113…  np.where(arr1>0) # it'll return indices(index) where data is greater than 0
          #return indices

Out[113…  (array([1, 2, 3, 4, 5, 9]),)
```

```
In [115…  np.extract(arr1>2, arr1) #extract  dataset which is equal to or True for given condition

Out[115…  array([324, 645,  65,   6,   6, 234])
```

numpy- Byte Swapping.

- represent data in internal byte order

```
In [116…  arr1

Out[116…  array([  0, 324, 645,  65,   6,   6,   0,   0,   0, 234])
```

```
In [119…  arr1.byteswap() # return data in byte, how it's stored inside system
          # passing True will update data in place

Out[119…  array([                  0,  4900197869555810304, -8862521116711714816,
           4683743612465315840,   432345564227567616,   432345564227567616,
       -1585267068834414592])
```

In [120…

```
arr1
```

Out[120…

```
array([  0, 324, 645,  65,   6,   6,   0,   0,   0, 234])
```

In [121…

```
arr1.byteswap(True)
```

Out[121…

```
array([                   0,  4900197869555810304, -8862521116711714816,
        4683743612465315840,   432345564227567616,   432345564227567616,
                          0,                    0,                    0,
       -1585267068834414592])
```

In [122…

```
arr1
```

Out[122…

```
array([                   0,  4900197869555810304, -8862521116711714816,
        4683743612465315840,   432345564227567616,   432345564227567616,
                          0,                    0,                    0,
       -1585267068834414592])
```

numpy- Copies & Views

- numpy.copy() ,create deep copy
- numpy.view() , create shallow copy

In [125…

```
arr1 = np.array([0,324,645,65,6,6,0,0,0,234])
```

In [126…

```
arr1
```

Out[126…

```
array([  0, 324, 645,  65,   6,   6,   0,   0,   0, 234])
```

In [127…

```
a = np.copy(arr1) # it'll create deep copy
```

In [128…

```
a
```

Out[128…

```
array([  0, 324, 645,  65,   6,   6,   0,   0,   0, 234])
```

In [138…

```
b = arr1.view() # create shallow copy
      #or
#b = arr1
#both are same
```

In [133…

```
b
```

```
[ ]:
```

```
[138]: b = arr1.view() # create shallow copy
             #or
       #b = arr1
       #both are same
```

```
[133]: b
```

```
[133]: array([  0, 324, 645,  65,   6,   6,   0,   0,   0, 234])
```

```
[134]: b[0] = 234
```

```
[135]: b
```

```
[135]: array([234, 324, 645,  65,   6,   6,   0,   0,   0, 234])
```

```
[136]: arr1
```

```
[136]: array([234, 324, 645,  65,   6,   6,   0,   0,   0, 234])
```

1 numpy- Matrix Library

```
[139]: import numpy.matlib as nm
```

```
[142]: nm.zeros(5) # matrix is subset of array so it perform same as array
```

```
[142]: matrix([[0., 0., 0., 0., 0.]])
```

```
[143]: nm.ones((3,4))
```

```
[143]: matrix([[1., 1., 1., 1.],
               [1., 1., 1., 1.],
               [1., 1., 1., 1.]])
```

```
[145]: nm.eye(4) # same as numpy.eye() , create identity matrix
```

```
[145]: matrix([[1., 0., 0., 0.],
               [0., 1., 0., 0.],
               [0., 0., 1., 0.],
               [0., 0., 0., 1.]])
```

2 numpy- Linear Algebra

```
[148]: arr1 = np.random.randint([[2,3] , [4,5]])
```

```
[149]: arr1
```

```
[149]: array([[1, 0],
              [2, 3]])
```

```
[ ]:
```

```
[152]: arr2 = np.random.randint([[5,3] , [2,5]])
```

```
[153]: arr2
```

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

2.0.1 matrix multiplication

```
[156]: np.dot(arr1,arr2)
```

```
[156]: array([[ 2,  1],
              [ 7, 11]])
```

```
[157]: arr1@arr2
```

```
[157]: array([[ 2,  1],
              [ 7, 11]])
```

```
[ ]:
```