In [1]:

```
import numpy as np
```

In [2]:

```
range(10)
```

Out[2]:

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

In [3]:

```
list(range(10))
```

Out[3]:

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

In [4]:

```
list(range(3,10))
```

Out[4]:

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

In [5]:

```
# With range function we can only get the integer values.

list(range(3,10.5))
```

```
---------------------------------------------------------------------------
-
TypeError                                 Traceback (most recent call las
t)
Input In [5], in <cell line: 1>()
----> 1 list(range(3,10.5))

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

In [6]:

```
# The advance version of 'range' is 'arange', by using this function we can get the valu

np.arange(10)
```

Out[6]:

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

```
start : [optional] start of interval range. By default start = 0
stop  : end of interval range
step  : [optional] step size of interval. By default step size = 1,
```

```
        For any output out, this is the distance between two adjacent values, out[i+1]
- out[i].
dtype : type of output array
```

In [7]:

```
np.arange(.5,10)
```

Out[7]:

```
array([0.5, 1.5, 2.5, 3.5, 4.5, 5.5, 6.5, 7.5, 8.5, 9.5])
```

In [8]:

```
np.arange(5 ,-4, -1)
```

Out[8]:

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

In [9]:

```
np.arange(5,-4,-.5)
```

Out[9]:

```
array([ 5. ,  4.5,  4. ,  3.5,  3. ,  2.5,  2. ,  1.5,  1. ,  0.5,  0. ,
       -0.5, -1. , -1.5, -2. , -2.5, -3. , -3.5])
```

```
1) The numpy.linspace() function returns number spaces with respect to interval.
2) Similar to numpy.arange() function but instead of step it uses sample number.

parameters :-
            start  : [optional] start of interval range. By default start = 0
            stop   : end of interval range
            restep : If True, return (samples, step). By default restep = False
            num    : [int, optional] No. of samples to generate
            dtype  : type of output array
```

In [11]:

```
# Simply we can say that 'linspace' will take " 1 to 5 Cm"  scale and divides it into th

np.linspace(1,5,20)
```

Out[11]:

```
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.        ])
```

In [14]:

```
list(np.linspace(1,5,20,retstep=True))
```

Out[14]:

```
[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.        ]),
 0.21052631578947367]
```

```
1) It also works like 'linspace' but the difference is 'logspace' will gives the
     logarithm of all the output values..shown below.(by default base = 10 )

parameters :-
             start    : [float] start(base ** start) of interval range.
             stop     : [float] end(base ** stop) of interval range
             endpoint : [boolean, optional]If True, stop is the last sample. By
default, True
             num      : [int, optional] No. of samples to generate
             base     : [float, optional] Base of log scale. By default, equals 10.0
             dtype    : type of output array
```

In [15]:

```
np.logspace(1,50,10)
```

Out[15]:

```
array([1.00000000e+01, 2.78255940e+06, 7.74263683e+11, 2.15443469e+17,
       5.99484250e+22, 1.66810054e+28, 4.64158883e+33, 1.29154967e+39,
       3.59381366e+44, 1.00000000e+50])
```

In [16]:

```
# 1) The numpy.zeros() function returns a new array of given shape and type, with zeros.

np.zeros(5)
```

Out[16]:

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

In [18]:

```
np.zeros((3,5))
```

Out[18]:

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

In [19]:

```
np.zeros((3,5,2))
```

Out[19]:

```
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.]]])
```

In [20]:

```
# Return a new array of given shape and type, filled with ones.

np.ones(5)
```

Out[20]:

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

In [21]:

```
# We can add (or) else we can any arithmetic operations for both 'one' & 'Zero'

np.ones((3,4,2)) + 5
```

Out[21]:

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

       [[6., 6.],
        [6., 6.],
        [6., 6.],
        [6., 6.]],

       [[6., 6.],
        [6., 6.],
        [6., 6.],
        [6., 6.]]])
```

In [22]:

```
np.ones((3,4,2)) * 5
```

Out[22]:

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

       [[5., 5.],
        [5., 5.],
        [5., 5.],
        [5., 5.]],

       [[5., 5.],
        [5., 5.],
        [5., 5.],
        [5., 5.]]])
```

In [23]:

```
# empty() :- Return a new array of given shape and type, without initializing entries.(M

np.empty((2,3))
```

Out[23]:

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

In [24]:

```
# eye() :- Return a 2-D array with ones on the diagonal and zeros elsewhere... as shown 

# simply we can say that it prints--- '1'   in the diagonal

np.eye(4)
```

Out[24]:

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

In [2]:

```
a = np.eye(4)
```

In [3]:

```
# This function tells that how many rows and columns inside the particular 'arrays' that

a.shape
```

Out[3]:

```
(4, 4)
```

In [4]:

```
a
```

Out[4]:

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

In [5]:

```
# This says that how many elements are there in a particular array.

a.size
```

Out[5]:

```
16
```

In [6]:

```
# This says the dimensions of the array

a.ndim
```

Out[6]:

```
2
```

In [7]:

```
a1 = np.random.randn(3,4)
```

In [8]:

```
a1
```

Out[8]:

```
array([[-0.48563886,  0.2517609 , -0.2528401 , -1.40609268],
       [-0.28325953,  1.27693428, -1.02328016, -0.26566824],
       [ 0.72122947, -0.8995494 ,  0.75406634, -1.69261206]])
```

In [9]:

```
# To extract the values from the array

a1[1][1]
```

Out[9]:

```
1.276934277716073
```

In [10]:

```
# To extract multiple values from the array

a1[0:2 , 0:2]
```

Out[10]:

```
array([[-0.48563886,  0.2517609 ],
       [-0.28325953,  1.27693428]])
```

In [11]:

```
# another way to extract the multiple values

a1[[0,1] , 0:2]
```

Out[11]:

```
array([[-0.48563886,  0.2517609 ],
       [-0.28325953,  1.27693428]])
```

In [12]:

```
a1
```

Out[12]:

```
array([[-0.48563886,  0.2517609 , -0.2528401 , -1.40609268],
       [-0.28325953,  1.27693428, -1.02328016, -0.26566824],
       [ 0.72122947, -0.8995494 ,  0.75406634, -1.69261206]])
```

In [13]:

```
# Extracting the data from '3rd column' and '0 & 1 rows'

a1[[1,2],3]
```

Out[13]:

```
array([-0.26566824, -1.69261206])
```

In [15]:

```
m1 = np.random.randint(1,3, (3,3))
```

In [16]:

```
m1
```

Out[16]:

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

In [17]:

```
m2 = np.random.randint(2,4, (3,3))
```

In [18]:

```
m2
```

Out[18]:

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

In [20]:

```
# Here it will not do the matrix multiplication.
# It will do the element multiplication.

m1 * m2
```

Out[20]:

```
array([[2, 6, 4],
       [2, 4, 4],
       [3, 3, 3]])
```

In [22]:

```
# Here it will do the matrix multiplication for this we will use the symbol called "@"

m1 @ m2
```

Out[22]:

```
array([[12, 13, 12],
       [12, 13, 12],
       [ 7,  8,  7]])
```

In [23]:

```
m1
```

Out[23]:

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

In [24]:

```
# This function will give the 'power' of each element in the 'm1'

pow(m1 , 4)
```

Out[24]:

```
array([[ 1, 16, 16],
       [ 1, 16, 16],
       [ 1,  1,  1]], dtype=int32)
```

In [25]:

```
# This function will give the 'square root' of each element in the 'm1'

np.sqrt(m1)
```

Out[25]:

```
array([[1.        , 1.41421356, 1.41421356],
       [1.        , 1.41421356, 1.41421356],
       [1.        , 1.        , 1.        ]])
```

In [26]:

```
# By using this function it will give the logarithmic of the 'm1'

np.log(m1)
```

Out[26]:

```
array([[0.        , 0.69314718, 0.69314718],
       [0.        , 0.69314718, 0.69314718],
       [0.        , 0.        , 0.        ]])
```

In [27]:

```
# For base-10 there is another function as shown below

np.log10(m1)
```

Out[27]:

```
array([[0.     , 0.30103, 0.30103],
       [0.     , 0.30103, 0.30103],
       [0.     , 0.     , 0.     ]])
```

In [32]:

```
# This function will give the 'exponential' of the 'm1'...as shown below

np.exp(m1)
```

Out[32]:

```
array([[2.71828183, 7.3890561 , 7.3890561 ],
       [2.71828183, 7.3890561 , 7.3890561 ],
       [2.71828183, 2.71828183, 2.71828183]])
```

In [33]:

```
# This method also gives the multiplication of each element.

m1**2
```

Out[33]:

```
array([[1, 4, 4],
       [1, 4, 4],
       [1, 1, 1]], dtype=int32)
```

In [ ]: