Numpy

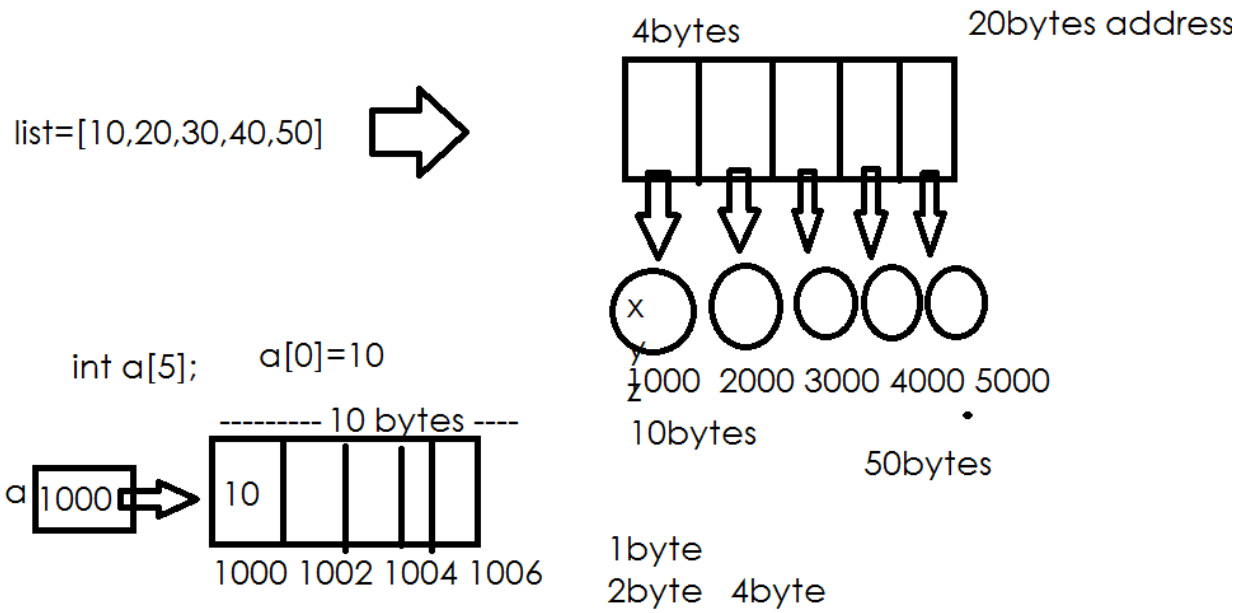
Numpy and pandas are data science libraries or api (application programming interface).

What is numpy?
Numpy stands for Numerical Python
Numpy is python library used for working with arrays (Vectors and Matrices).
Numpy provide functions to work with linear algebra, fourier transform and matrices.
Numpy is a python library is partially written in python and major part is wrriten in **C language.**

Array is collection of similar type data or elements.

Q: What is difference between Array and List?

List	**Array**
List is a collection of heterogeneous data elements.	Array is collection of homogeneous data elements.
List does not have axis or dimensions.	Array is having axis or dimensions
In List data will stored as objects.	In array data will stored as scalar value.
List is dynamic in size.	Array is fixed in size.
Because of list store data as objects it occupy more space.	Because of array store data as scalar values it occupy less space.
It is not efficient to process large sets of data.	It is efficient to process large set of data.
List hold only objects types.	Array holds scalar types and objects types.



pip install numpy

(OR)
Download and install anaconda distribution
Anaconda is a python distribution which provides,

1. **Python softwrare**
2. **Data Science and ML Libraries**
    a. **Numpy**
    b. **Pandas**
    c. **Matplotlib**
    d. **Scipy**
    e. **TersonFlow**
    f. **ScikitLearn**
3. **IDE's (Jupiter, Spyder)**

https://www.anaconda.com/products/individual

Jupyter notebook
Jupyter notebook is web application/internet application
The ***Jupyter Notebook*** is a web-based interactive computing platform. The notebook combines live code, equations, narrative text, visualizations, ...

Creating numpy array

Numpy is used to create arrays. The array object in numpy is called ndarray (OR) ndarray is data type or class which represent array object. ndarray class is used to create array objects in python.
Numpy provide various functions for creating array.

Numpy provides various functions for creating array object.

1. array()
2. arange()
3. zeros()
4. ones()
5. full()
6. eye()
7. identity()

1. array() : The function create ndarray object.

dtype: dtype is attribute of array, which define data type, the default type of array is object.

Numpy provide the following data types.

1. Integer data types.
    a. Int8 → 1byte
    b. Int16 → 2bytes
    c. Int32 → 4bytes
    d. Int64 → 8bytes
2. Float data types
    a. Float16 → 2bytes
    b. Float32 → 4bytes
    c. Float64 → 8bytes
3. Complex data types
    a. Complex64 → 8bytes
    b. Complex128 → 16bytes
4. Unsigned int data type
    a. Uint8 → 1byte
    b. Uint16 → 2bytes
    c. Uint32 → 4bytes
    d. Uint64 → 8bytes

This data types also represented using single characters.

1. i→ integer
2. f → float
3. u → unsigned int
4. s → string

Syntax of array() function:
array(object,dtype,order,ndim)

object: object is an iterable/sequence which is used to construct array.
dtype: this indicates datatype of array
order: order can be C (row-major), F(column-major)
ndim: This specifies minimum number of dimensions of an output array.

Creating numpy array
Numpy is used to create arrays. The array object in numpy is called ndarray.
ndarray class is used to create array objects in python.
Numpy provide various functions for creating array.

2. array() : The function create ndarray object.

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

Numpy moudle impored as alias name np

```
a=np.array([10,20,30,40,50])
print(a)
print(type(a))
b=np.array([[1,2,3],[4,5,6],[7,8,9]])
print(b)
```

```
[10 20 30 40 50]
<class 'numpy.ndarray'>
[[1 2 3]
 [4 5 6]
 [7 8 9]]
```

---

dtype: dtype is attribute of array, which define data type, the default type of array object.
Numpy provide the following data types.

5. Integer data types.
    a. Int8 → 1byte
    b. Int16 → 2bytes
    c. Int32 → 4bytes
    d. Int64 → 8bytes
6. Float data types
    a. Float16 → 2bytes
    b. Float32 → 4bytes
    c. Float64 → 8bytes
7. Complex data types
    a. Complex64 → 8bytes
    b. Complex128 → 16bytes
8. Unsigned int data type
    a. Uint8 → 1byte
    b. Uint16 → 2bytes
    c. Uint32 → 4bytes
    d. Uint64 → 8bytes

This data types also represented using single characters.

5. i→ integer
6. f → float
7. u → unsigned int
8. s → string

Syntax of array() function:

array(object,dtype,order,ndim)

object: object is an iterable/sequence which is used to construct array.
dtype: this indicates datatype of array
order: order can be C (row-major), F(column-major)
ndim: This specifies minimum number of dimensions of an output array.

```
[7] a=np.array([10,20,30,40,50],dtype=np.int8)
    print(a.dtype)
    print(a.ndim)
    print(a.shape)
    print(a)
```

```
int8
1
(5,)
[10 20 30 40 50]
```

```
[12] b=np.array([10,20,30,40,50,'python'])
     print(b)
     print(b.dtype)
```

```
['10' '20' '30' '40' '50' 'python']
<U21
```

```
[15] c=np.array([10,20,30,40,50,'60'],dtype=np.int8)
     print(c)
```

```
[10 20 30 40 50 60]
```

```
d=np.array([10,20,30,40,50],dtype=np.float16)
print(d)
```

```
[10. 20. 30. 40. 50.]
```

Q: What is dimension?
Dimension define depth of array or nested.
Q: What is shape of the array/
Shape define the number of rows and columns
Q: What is size?
The size will define total number elements exists within array
Q: What is dtype?
Type of elements hold by array

```
a=np.array([[1,2,3],[4,5,6],[7,8,9]])
print(a.ndim) # dimension define depth of array
print(a.shape) # number of rows and columns
print(a.dtype) # type of data hold by array
print(a.size) # total number of element exists in array
print(a.itemsize) # return size of element
print(a[0][0],a[0][1],a[0][2])
print(a[1][0],a[1][1],a[1][2])
print(a[2][0],a[2][1],a[2][2])
```

```
2
(3, 3)
int64
9
8
1 2 3
4 5 6
7 8 9
```

```
a=np.array([[1,2,3],[4,5,6],[7,8,9]])
print(a[0][0],a[0][1],a[0][2])
print(a[1][0],a[1][1],a[1][2])
print(a[2][0],a[2][1],a[2][2])
b=np.array([[1,2,3],[4,5,6],[7,8,9]],order='F')
print(b[0][0],b[0][1],b[0][2])
print(b[1][0],b[1][1],b[1][2])
print(b[2][0],b[2][1],b[2][2])
```

```
1 2 3
4 5 6
7 8 9
1 2 3
4 5 6
7 8 9
```

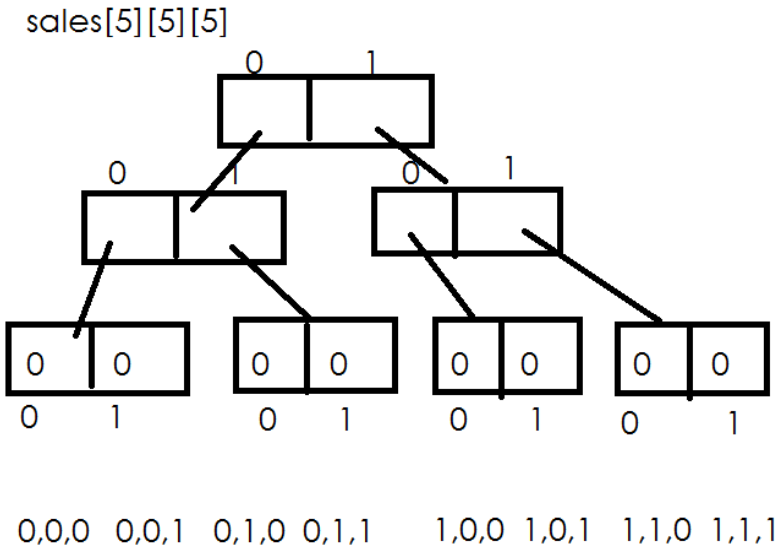
Other functions of creating numpy array

1. **empty()**
   This function return empty array.
   Syntax:
   empty(shape,dtype=float)

```
import numpy as np
a=np.empty(shape=(3,))
print(a)
b=np.empty(shape=(2,3))
print(b)
c=np.empty(shape=(10,),dtype=np.int)
print(c)
```

```
[0.75 0.75 0.  ]
[[4.66341068e-310 0.00000000e+000 0.00000000e+000]
 [0.00000000e+000 0.00000000e+000 0.00000000e+000]]
[94388476715408   476741369968   498216206450   472446402592
   468151435381   519691042928   416611827744   137438953587
   481036337262              0]
```

This will return ndarray with uninitialized values.



2. **zeros()**
   This function return array with zero filled values.
   Syntax:
   zeros(shape,dtype=float)

```
a=np.zeros(shape=(5,))
print(a)
b=np.zeros(shape=(3,3),dtype=np.int16)
print(b)
print(b.itemsize)
```

```
[0. 0. 0. 0. 0.]
[[0 0 0]
 [0 0 0]
 [0 0 0]]
2
```

3. **ones()**
   This function return array with ones filled values.
   Syntax:
   ones(shape,dtype=float)

```
a=np.ones(shape=(5,))
print(a)
b=np.ones(shape=(4,4),dtype=np.int8)
print(b)
```

```
[1. 1. 1. 1. 1.]
[[1 1 1 1]
 [1 1 1 1]
 [1 1 1 1]
 [1 1 1 1]]
```

4. asarray()
   This function return array object (OR) this function is used to convert input object into array object.
   Syntax:
   asarray(object,dtype=None)
   If the data type is not defined, it takes the data types based input object elements types.

```
a=np.ones(shape=(3,3),dtype=np.int8)
print(a)
b=np.asarray(a)
print(b)
```

```
[[1 1 1]
 [1 1 1]
 [1 1 1]]
[[1 1 1]
 [1 1 1]
 [1 1 1]]
```

5. frombuffer()
   Buffer is temp storage memory area
   Buffers are used to increase efficiency in read and writing.
   frombuffer() function is used to construct array object using buffer.
   Buffer is a collection of bytes

   Syntax:
   frombuffer(buffer,dtype=float,count=-1,offset=0)

```
buf=b'NARESHIT'
print(type(buf))
a=np.frombuffer(buf,dtype="S1")
print(a)
b=np.frombuffer(buf,dtype="S1",count=3)
print(b)
c=np.frombuffer(buf,dtype="S1",count=2,offset=3)
print(c)
```

```
<class 'bytes'>
[b'N' b'A' b'R' b'E' b'S' b'H' b'I' b'T']
[b'N' b'A' b'R']
[b'E' b'S']
```

6. fromiter()
   This function return numpy array object using iterator object.
   Syntax:
   fromiter(iterable,dtype,count=-1)
   iterable that will provide data for one dimension array object
   count represents the number of elements should read from iterable, the default is -1 which indicates all elements.

```
g=(n for n in range(1,11))
print(g)
a=np.fromiter(g,dtype=np.int8)
print(a)
g1=(n for n in range(1,11) if n%2!=0)
b=np.fromiter(g1,dtype=np.int8)
print(b)
list1=list(range(1,21))
c=np.fromiter(list1,dtype=np.int8)
print(c)
```

```
<generator object <genexpr> at 0x7f234e9c9d50>
[ 1  2  3  4  5  6  7  8  9 10]
[1 3 5 7 9]
[ 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20]
```

7. **arange()**
   The function return array object with numeric range of values.
   Syntax:
   arange(start,stop,step,dtype=None)

   start: start value of the array
   stop: end value of the array, which is not included

step: difference between two values or increment or decrement value, the default value is 1

```
a=np.arange(start=1,stop=11,step=2)
print(a)
b=np.arange(start=0,stop=21,step=2)
print(b)
```

```
[1 3 5 7 9]
[ 0  2  4  6  8 10 12 14 16 18 20]
```

8. linspace()
   this function works like arange function.
   This function takes size of array. It constructs the array based on number of elements. The difference between each element (step) it taken based on number of elements.

```
import numpy as np
a=np.linspace(1,5,5)
print(a)
b=np.linspace(1,5,10)
print(b)
c=np.linspace(1,5,7,dtype=np.int8)
print(c)
```

```
[1. 2. 3. 4. 5.]
[1.         1.44444444 1.88888889 2.33333333 2.77777778 3.22222222
 3.66666667 4.11111111 4.55555556 5.        ]
[1 1 2 3 3 4 5]
```

9. **logspace()**
   The logspace function create numpy array object evenly separated values using log scale.
   Syntax:
   logspace(start,stop,num,endpoint,base,dtype)

   start: start specifies the start value of the interval
   stop: specifies the stop value of/end value of the interval
   num: The number of values to be generated within range
   endpoint: The value of endpoint can be True or False, if endpoint is True, which tells logspace include stop value, if false exclude stop value.
   base: represent base of the log space
   dtype: the type of data hold by numpy array.

```
a=np.logspace(1,50)
print(a)
b=np.logspace(10,20,5)
print(b)
```

```
[1.e+01 1.e+02 1.e+03 1.e+04 1.e+05 1.e+06 1.e+07 1.e+08 1.e+09 1.e+10
 1.e+11 1.e+12 1.e+13 1.e+14 1.e+15 1.e+16 1.e+17 1.e+18 1.e+19 1.e+20
 1.e+21 1.e+22 1.e+23 1.e+24 1.e+25 1.e+26 1.e+27 1.e+28 1.e+29 1.e+30
 1.e+31 1.e+32 1.e+33 1.e+34 1.e+35 1.e+36 1.e+37 1.e+38 1.e+39 1.e+40
 1.e+41 1.e+42 1.e+43 1.e+44 1.e+45 1.e+46 1.e+47 1.e+48 1.e+49 1.e+50]
[1.00000000e+10 3.16227766e+12 1.00000000e+15 3.16227766e+17
 1.00000000e+20]
```

10. full()

numpy.full(shape, fill_value, dtype=None)
Return a new array of given shape and type, filled with *fill_value*.

```
In [21]: a=np.full((5,),fill_value=10)
         print(a)
         b=np.zeros(shape=(5,))
         print(b)
         c=np.full((3,3),fill_value=5)
         print(c)
```

```
[10 10 10 10 10]
[0. 0. 0. 0. 0.]
[[5 5 5]
 [5 5 5]
 [5 5 5]]
```


11. eye()

numpy.eye(N, M=None, k=0, dtype=<class 'float'>)
Return a 2-D array with ones on the diagonal and zeros elsewhere.

Nint
Number of rows in the output.
Mint, optional
Number of columns in the output. If None, defaults to N.
kint, optional
Index of the diagonal: 0 (the default) refers to the main diagonal, a positive value refers to an upper diagonal, and a negative value to a lower diagonal.
dtypedata-type, optional

Data-type of the returned array.

```
In [24]: a=np.eye(4)
         print(a)
         b=np.eye(5,k=1)
         print(b)
         c=np.eye(3,k=-1)
         print(c)
```

```
[[1. 0. 0. 0.]
 [0. 1. 0. 0.]
 [0. 0. 1. 0.]
 [0. 0. 0. 1.]]
[[0. 1. 0. 0. 0.]
 [0. 0. 1. 0. 0.]
 [0. 0. 0. 1. 0.]
 [0. 0. 0. 0. 1.]
 [0. 0. 0. 0. 0.]]
[[0. 0. 0.]
 [1. 0. 0.]
 [0. 1. 0.]]
```


12. Identity()

numpy.identity(*n*, *dtype=None*)

Return the identity array.

The identity array is a square array with ones on the main diagonal.

```
In [27]: a=np.identity(3)
         print(a)
         b=np.identity(5)
         print(b)
```

```
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
[[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.]]
```

How to read elements from Array using indexing and slicing

Using index we can read only one element

Slicing uses multiple indexes to read more than one value

Using this approach we can read elements of any dimension

```
a=np.array([[1,2,3],[4,5,6],[5,6,7]])
print(a.ndim)
print(a.shape)
print(a.size)
print(a[0],a[1],a[2])
print(a[0][0],a[0][1],a[0][2])
print(a[1][0],a[1][1],a[1][2])
print(a[2][0],a[2][1],a[2][2])
row,col=a.shape
for i in range(row):
  for j in range(col):
    print(a[i][j],end=' ')
  print()
```

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

```
a=np.array([10,20,30,40,50,60,70,80,90,100])
print(a)
b=a[0:4]
print(b)
print(type(b))
b=np.array([[1,2,3],[4,5,6],[7,8,9]])
c=b[0:2]
print(c)
```

```
[ 10  20  30  40  50  60  70  80  90 100]
[10 20 30 40]
<class 'numpy.ndarray'>
[[1 2 3]
 [4 5 6]]
```

Slicing on matrix:

matrix[row_lower:row_upper,col_lower:col_upper]

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

```
l=[[1,2,3,4],[5,6,7,8],[9,10,11,12],[13,14,15,16]]
a=np.array(l)
print(a)
b=a[0:2,0:2]
print(b)
c=a[1:,2:]
print(c)
d=a[:,:]
print(d)
e=a[:,0:2]
print(e)
```

```
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]
 [13 14 15 16]]
[[1 2]
 [5 6]]
[[ 7  8]
 [11 12]
 [15 16]]
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]
 [13 14 15 16]]
[[ 1  2]
 [ 5  6]
```

✓ 0s completed at 7:08 PM

Advanced Indexing

Using numpy advanced indexing we can read selected values from matrix using collection of tuples which represent row and col index. It allows to read even using condition.

Advanced indexing is classified into two categories.

1. Integer based index
2. Boolean based index

Integer based index

In integer based we try to create a collection which consist of tuples, where each tuple represent rowindex and colindex.

```
a=np.array([[1,2,3],[4,5,6],[7,8,9]])
print(a)
x=([0,1,1,2],[1,1,2,2])
b=a[x]
print(b)
```

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

Boolean based index

In this approach we defined boolean expression in indices.

If the expression return True, the value is return else value is not return from array.

```
import numpy as np
a=np.array([[1,2,3],[4,5,6],[7,8,9],[10,11,12]])
print(a)
print(a[a>5])
b=np.array([[10,4,7],[6,9,20],[4,2,9]])
c=b[b%2==0]
print(c)
```

```
[[ 1  2  3]
 [ 4  5  6]
 [ 7  8  9]
 [10 11 12]]
[ 6  7  8  9 10 11 12]
[10  4  6 20  4  2]
```

Iterating over array
Iteration or using for loop:
We can process the elements of array using for loop.
Using the looping statement, we can iterate the elements of array in forward direction.

```
a=np.array([10,20,30,40,50,60,70,80,90,10])
for value in a:
  print(value,end=' ')
b=np.array([[1,2,3],[4,5,6],[7,8,9]])
print()
for row in b:
  for col in row:
    print(col,end=' ')
  print()
```

```
10 20 30 40 50 60 70 80 90 10
1 2 3
4 5 6
7 8 9
```

Control the iteration order
Depending on application requirement, we can access or process the element in specific order.
The nditr() function return an iterator object, which control the order of elements.
We can define the order as row major order or column major order.
Which is represented using “C” or “F”.

```
a=np.arange(9) # (9,) --> 1-D
a=a.reshape(3,3) # 3x3     --> 2-D
print(a)
i=np.nditer(a,order='C')
for value in i:
  print(value)
j=np.nditer(a,order='F')
for value in j:
  print(value)
```

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

Using external loop

```
a=np.arange(9)
a=a.reshape(3,3)
for value in a:
  print(value)
for value in np.nditer(a,flags=['external_loop']):
  print(value)
for value in np.nditer(a,flags=['external_loop'],order='F'):
  print(value)
```

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

Array manipulations

reshape : This gives new shape to the array without modifying existing data. The shape define the number of rows and columns.

Syntax:

np.reshape(array,newshape,order='C')

```
a=np.arange(9)
print(a) # 1-D array with the shape of (9,)
b=np.reshape(a,(3,3)) # 2-D array with shape of (3,3)
print(b)
print(a.shape)
print(b.shape)
```

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

np.flat : one dimensional iterator over the array.

```
a=np.arange(1,7).reshape(2,3)
print(a)
print(a.flat[3])
print(a.flat[5])
```

```
[[1 2 3]
 [4 5 6]]
4
6
```

ndarray.flatten() : return one dimensional array

```
a=np.arange(1,7).reshape(2,3)
print(a)
b=a.flatten()
print(b)
```

```
[[1 2 3]
 [4 5 6]]
[1 2 3 4 5 6]
```

numpy.transpose()

This function/method returns transpose of matrix (OR) this method returns reverse axes of matrix.

This function receives the input as array and transpose and return modified array.

```
import numpy as np
matrix1=np.array([[1,2,3],[4,5,6],[7,8,9]])
print(matrix1)
matrix2=np.transpose(matrix1)
print(matrix2)
```

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

ndarray.T

T represents transposed array/matrix

This will return new matrix with changes.

It is member/method of ndarray.

```
import numpy as np
matrix1=np.array([[1,2,3],[4,5,6],[7,8,9]])
matrix2=matrix1.T
print(matrix1)
print(matrix2)
```

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

numpy.swapaxes(array,axis1,axis2)

This function is used to interchange axis of an array.

```
matrix1=np.array([[1,2,3]])
print(matrix1.shape)
print(matrix1)
matrix2=np.swapaxes(matrix1,0,1)
print(matrix2)
matrix3=np.array([[1,2,3],[4,5,6],[7,8,9]])
print(matrix3)
matrix4=np.swapaxes(matrix3,0,1)
print(matrix4)
matrix5=np.swapaxes(matrix3,1,0)
print(matrix5)
```

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

numpy.rollaxis()
rollaxis function will roll the specified axis in backward direction until specified position.
Syntax: numpy.rollaxis(a,axis,start=0)

```
a=np.ones((1,2,3,4),dtype=np.int8)
print(a.ndim)
print(a)
print(a.shape)
print(np.rollaxis(a,3,1).shape)
```

```
4
[[[[1 1 1 1]
   [1 1 1 1]
   [1 1 1 1]]

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

np.expand_dim(array,axis)
This function is used to expand or modify dimension of a specified array. We can insert new axis by defining the position.

```
a=np.array([1,2])
print(a.ndim)
print(a.shape)
print(a)
b=np.expand_dims(a,axis=0)
print(b.ndim)
print(b)
print(b.shape)
c=np.expand_dims(a,axis=1)
print(c.ndim)
print(c.shape)
print(c)
```

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

Arithmetic Operations on numpy array
numpy provides various functions to perform arithmetic operations.
The following methods provided by ndarray.

1. add()
2. subtract()
3. multiply()
4. divide()
5. dot()
6. floor_divide()
7. mod()
8. power()

All above operations can be done using operators.
Example: add() - + , subtract() -

add() : This function add two arrays

```
>>> a1=np.array([1,2,3])
>>> a2=np.array([4,5,6])
```

```
>>> a3=np.add(a1,a2)
>>> print(a1,a2,a3,sep="\n")
[1 2 3]
[4 5 6]
[5 7 9]
>>> m1=np.array([[1,2,3],
...              [4,5,6],
...              [7,8,9]])
>>> m2=np.array([[3,4,5],
...              [9,8,7],
...              [3,2,1]])
>>> m3=m1+m2
>>> print(m1,m2,m3,sep="\n")
[[1 2 3]
 [4 5 6]
 [7 8 9]]
[[3 4 5]
 [9 8 7]
 [3 2 1]]
[[ 4  6  8]
 [13 13 13]
 [10 10 10]]
>>>
```

numpy.subtract(x1,x2)
Subtract arguments, element-wise.

```
>>> a=np.array([4,5,6])
>>> b=np.array([1,2,3])
>>> c=np.subtract(a,b)
>>> print(a,b,c,sep="\n")
[4 5 6]
[1 2 3]
[3 3 3]
>>> x=np.array([[4,5,6],
...             [8,9,10]])
>>> y=np.array([[1,2,3],
...             [4,5,6]])
>>> z=np.subtract(x,y)
>>> print(x,y,z,sep="\n")
```

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

numpy.multiply(x1,x2)
Multiply arguments element-wise.

```
>>> a=np.array([4,5,6])
>>> b=np.array([1,2,3])
>>> c=np.multiply(a,b)
>>> print(a,b,c,sep="\n")
[4 5 6]
[1 2 3]
[ 4 10 18]
>>> x=np.array([[4,5,6],
...             [8,9,10]])
>>> y=np.array([[1,2,3],
...             [4,5,6]])
>>> z=np.multiply(x,y)
>>> print(x,y,z,sep="\n")
[[ 4  5  6]
 [ 8  9 10]]
[[1 2 3]
 [4 5 6]]
[[ 4 10 18]
 [32 45 60]]
```

numpy.dot(a,b)
Dot product of two arrays

```
>>> m1=np.array([[1,2],
...              [3,4]])
>>> m2=np.array([[4,5],
...              [6,7]])
```

```
>>> m3=np.dot(m1,m2)
>>> print(m1,m2,m3,sep="\n")
[[1 2]
 [3 4]]
[[4 5]
 [6 7]]
[[16 19]
 [36 43]]
>>>
```

numpy.divide(x1,x2)
Divide arguments element-wise.

```
>>> a=np.array([4,5,6])
>>> b=np.array([1,2,3])
>>> c=np.divide(a,b)
>>> print(a,b,c,sep="\n")
[4 5 6]
[1 2 3]
[4.  2.5 2. ]
```

numpy.floor_divide(x1,x2)
Divide arguments element-wise. Return result in integer

```
>>> a=np.array([4,5,6])
>>> b=np.array([1,2,3])
>>> d=np.floor_divide(a,b)
>>> print(a,b,d,sep="\n")
[4 5 6]
[1 2 3]
[4 2 2]
>>>
```

numpy.mod(*x1*, *x2*)
Returns the element-wise remainder of division.

```
>>> a=np.array([4,5,6])
>>> b=np.array([1,2,3])
>>> c=np.mod(a,b)
>>> print(a,b,c,sep="\n")
[4 5 6]
```

```
[1 2 3]
[0 1 0]
>>>
```

numpy.power(x1,x2)
First array elements raised to powers from second array, element-wise.

```
>>> a=np.array([4,5,6])
>>> b=np.array([1,2,3])
>>> c=np.power(a,b)
>>> print(a,b,c,sep="\n")
[4 5 6]
[1 2 3]
[  4  25 216]
```

Statistical Operations
Numpy library provides the functions to perform statistical operations on array.

amax()
amin()
mean()
median()
var()
std()

amax()
Return the maximum of an array or maximum along an axis.
numpy.amax(*a*, *axis=None)*

```
>>> a=np.array([10,20,30,40,50])
>>> np.amax(a)
50
>>> b=np.array([[1,2,3],
...             [4,5,6],
...             [7,8,9]])
>>> np.amax(b)
9
>>> np.amax(b,axis=0)
```

```
array([7, 8, 9])
>>> np.amax(b,axis=1)
array([3, 6, 9])
```

amin()

numpy.**amin**(*a*, *axis=None)*
Return the minimum of an array or minimum along an axis.

```
>>> a=np.array([10,20,30,40,50])
>>> np.amin(a)
10
>>> b=np.array([[1,2,3],
...             [4,5,6],
...             [7,8,9]])
>>> np.amin(b)
1
>>> np.amin(b,axis=0)
array([1, 2, 3])
>>> np.amin(b,axis=1)
array([1, 4, 7])
>>>
```

mean()

numpy.**mean**(*a*, *axis=None*)
Compute the arithmetic mean along the specified axis.
Returns the average of the array elements. The average is taken over the flattened array by default, otherwise over the specified axis. float64 intermediate and return values are used for integer inputs.

```
a=np.array([1,2,3,4,5,6,7,8,9,10])
print(a)
[ 1  2  3  4  5  6  7  8  9 10]
>>> m=np.mean(a)
>>> print(m)
5.5
>>> b=np.array([[1,2,3],
...             [4,5,6],
```

```
...             [7,8,9]])
>>> print(b)
[[1 2 3]
 [4 5 6]
 [7 8 9]]
>>> m=np.mean(b)
>>> print(m)
5.0
>>> m1=np.mean(b,axis=0)
>>> print(m1)
[4. 5. 6.]
>>> m2=np.mean(b,axis=1)
>>> print(m2)
[2. 5. 8.]
>>>
```

median()

numpy.**median**(*a*, *axis=None*)

Compute the median along the specified axis.

Returns the median of the array elements.

This function returns middle element

Sort the elements of array in ascending order

If the number of elements in array is odd number, it returns middle element

If the number of elements in array is even number, it read two middle elements, add it and divide with 2

```
>>> a=np.array([1,2,3,4,5,6,7])
>>> print(a)
[1 2 3 4 5 6 7]
>>> np.median(a)
4.0
>>> b=np.array([1,2,3,4,5,6,7,8])
>>> print(b)
[1 2 3 4 5 6 7 8]
>>> np.median(b)
4.5
```

var()

numpy.**var**(*a, axis=None)*
Compute the variance along the specified axis.
Returns the variance of the array elements, a measure of the spread of a distribution. The variance is computed for the flattened array by default, otherwise over the specified axis.
variance=sum(sqr(mean-xi))/total of number of elements

Xi is an each element of array

```
>>> a=np.array([1,2,3,4,5])
>>> print(a)
[1 2 3 4 5]
>>> np.var(a)
2.0
```

std()
numpy.**std**(*a, axis=None)*
Compute the standard deviation along the specified axis.
Returns the standard deviation, a measure of the spread of a distribution, of the array elements. The standard deviation is computed for the flattened array by default, otherwise over the specified axis.

std=sqrt(var)

```
>>> a=np.array([1,2,3,4,5])
>>> print(a)
[1 2 3 4 5]
>>> np.var(a)
2.0
>>> np.std(a)
1.4142135623730951
>>> b=np.array([1,2,3,4,5,6,7,8,9,10])
>>> np.var(b)
8.25
>>> np.std(b)
2.8722813232690143
>>> import math
```

```
>>> math.sqrt(np.var(b))
2.8722813232690143
```