numpy :

numpy is known as numeric python.

numpy is :

- it stands for numerical python
- provide nD-array and logical or mathematical operations on array
- linear algebric expression on array/metrix.

use of numeric python -

- it is used for working with arrays
- it is easy to use
- easy to maintain
- provide better and easier way to handle the data
- to work with sufficiant memory allocation

commands :

- to install numpy - pip install numpy
- to update numpy - pip install --upgrade numpy

to check it/use it in code:

- import pkg - import numpy
- after it we can use it as np - import numpy as np
- check numpy version - numpy.__ version __

Combination of numpy with other packages:

- It can be used with other packages like Scipy(Scientific Python) and Mat-plotlib(Plotting library).
- Numpy is open–source and can be used as a replacement for matlab.

command to import or install numpy using cmd mode:

```
pip install numpy

Requirement already satisfied: numpy in c:\users\rajen\anaconda3\envs\
notebook\lib\site-packages (1.26.4)
Note: you may need to restart the kernel to use updated packages.

pip install --upgrade numpy

Requirement already satisfied: numpy in c:\users\rajen\anaconda3\envs\
notebook\lib\site-packages (1.26.4)
Note: you may need to restart the kernel to use updated packages.
```

```
import numpy
```

check numpy version using code :

```
import numpy

print(numpy.__version__)
```

```
1.26.4
```

basic functions on numpy array

Let's see numpy by an example :

```
import numpy

x = numpy.array([1,2,3,4,5])
print(x)
```

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

- use numpy as np

```
import numpy as np

x = np.array([1,3,5,7,9])
print(x)
```

```
[1 3 5 7 9]
```

- type of array

```
import numpy as np

x = np.array([1,3,5,7,9])
print(type(x))
```

```
<class 'numpy.ndarray'>
```

- length of array

```
import numpy as np

x = np.array([1,3,5,7,9])
print(len(x))
```

```
5
```

- size of array in bytes

```
import numpy as np

x = np.array([1,3,5,7,9])
print(x.nbytes)
```

```
20
```

- in above array is collection of 5 integer and eeach int hold 2 byte so size of array in byte is : 20
- size of array / total number of elements in array

```
import numpy as np

x = np.array([1,3,5,7,9])
print(x.size)
```

```
5
```

```
import numpy as np

x = np.array([[1,2,3],[5,6,7]])
print(x.size)
```

```
6
```

- shape of array (number of rows , numbers of columns)

```
import numpy as np

x = np.array([1,3,5,7,9])
print(x.shape)
```

```
(5,)
```

```
import numpy as np

x = np.array([[1,2,3],[5,6,7]])
print(x.shape)
```

```
(2, 3)
```

NOTE : Integers at every index tells about the number of elements the corresponding dimension has, In the example above at index-4 we have value 4, so we can say that 5th (4 + 1 th) dimension has 4 elements.

- you can write code in diffeent-diffeent cells :

```
import numpy as np

ar = np.array([1,2,3,4,5])

ar
```

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

```
len(ar)
```

```
5
```

```
type(ar)
```

```
numpy.ndarray
```

Numpy nD-Array:

- Important array object defined in Numpy is an n-dimensional array type called ndarray.
- It describes the group of items of the same type.
- Each item in the collection can be accessed using the zero-based index.
- Every item in the ndarray occupies the same amount of memory.
- Each element in an ndarray is an object of data-type object(called dtype).
- numpy array/ python array : it is a collection of similar or dissimilar elements. there are 4 types of array:

1. 0D-ARRAY
2. 1D-ARRAY
3. 2D-ARRAY
4. nD-ARRAY OR 3D ARRAY

basic example of 0D, 1D, 2D and nD ARRAY...

1. 0D-Array : 0-D arrays, or Scalars, are the elements in an array. Each value in an array is a 0-D array.

Create a 0-D array with value 420:

```
import numpy as np

arr = np.array(420)

print(arr)
print(type(arr))
```

```
420
<class 'numpy.ndarray'>
```

2. 1D-Array : An array that has 0-D arrays as its elements is called uni-dimensional or 1-D array. These are the most common and basic arrays.

Create a 1-D array containing the values 1,2,3,4,5:

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5])
```

```
print(arr)
print(type(arr))
```

```
[1 2 3 4 5]
<class 'numpy.ndarray'>
```

3. 2D-Array : An array that has 1-D arrays as its elements is called a 2-D array. These are often used to represent matrix or 2nd order tensors.

note : NumPy has a whole sub module dedicated towards matrix operations called numpy.mat

Create a 2-D array containing two arrays with the values 1,2,3 and 4,5,6:

```
import numpy as np

arr = np.array([[1, 2, 3],[4,5,6]])

print(arr)
print(type(arr))
```

```
[[1 2 3]
 [4 5 6]]
<class 'numpy.ndarray'>
```

4. nD-Array or 3D-Array : An array that has 2-D arrays (matrices) as its elements is called 3-D array. These are often used to represent a 3rd order tensor.

Create a 3-D array with two 2-D arrays, both containing two arrays with the values 1,2,3 and 4,5,6:

```
import numpy as np

arr = np.array([[[1, 2, 3], [4, 5, 6]], [[1, 2, 3], [4, 5, 6]]])

print(arr)
print(type(arr))
```

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

 [[1 2 3]
  [4 5 6]]]
<class 'numpy.ndarray'>
```

SYNYAX : array_name = numpy.array(object,dtype=None,copy=True,order=None,subok=False,ndmin=0) parameters in numpy array :

- Object – Any object exposing the array interface method returns an array object.
- Dtype – Desired datatype of the object.

- Copy – By default,the object is copied.
- Order - C (row major) or F (column major) or A (any) (default).
- Subok – Returned array is forced to be a base class array.
- Ndmin – Specifies min dimensions of resultant array.

Data Types in NumPy

NumPy has some extra data types, and refer to data types with one character, like i for integers, u for unsigned integers etc. Below is a list of all data types in NumPy and the characters used to represent them. i - integer b - boolean u - unsigned integer f - float c - complex float m - timedelta M - datetime O - object S - string U - unicode string V - fixed chunk of memory for other type (void)

- ***parameters in dtype/ data type :*** dtype stands for data type of array elements. It supports a much greater variety of numerical types than python does. A dtype object is constructed.

syntax: Numpy.dtype(object,align,copy)

parameter :

- Object – to be converted to datatype object.
- Align – If true , adds padding to the field.
- Copy – If true , makes a new copy of the dtype object.

Example :

```
import numpy as np

arr = np.array([1, 2, 3, 4])
print(arr.dtype)
```

```
int32
```

```
import numpy as np

arr = np.array([1j, 2+3j, 3, 4])
print(arr.dtype)
```

```
complex128
```

```
import numpy as np

arr = np.array(['a','s','d','g'])
print(arr.dtype)
```

```
<U1
```

```
import numpy as np

arr = np.array(['apple', 'banana', 'cherry'])
print(arr.dtype)
```

```
<U6
```

```
import numpy as np

a = np.array([1, 2, 3], dtype = complex)
print(a)
```

```
[1.+0.j 2.+0.j 3.+0.j]
```

- For i, u, f, S and U we can define size as well.

```
import numpy as np

arr = np.array([1, 2, 3, 4], dtype='S')
print(arr)
print(arr.dtype)
```

```
[b'1' b'2' b'3' b'4']
|S1
```

```
import numpy as np

arr = np.array([1, 2, 3, 4], dtype='i4')
print(arr)
print(arr.dtype)
```

```
[1 2 3 4]
int32
```

- using array-scalar type:

```
import numpy as np

dt = np.dtype(np.int32)
print(dt)
```

```
int32
```

ERORR:

What if a Value Can Not Be Converted? If a type is given in which elements can't be casted then NumPy will raise a ValueError. *ValueError: In Python ValueError is raised when the type of passed argument to a function is unexpected/incorrect.* ***Converting Data Type on Existing Arrays*** :The best way to change the data type of an existing array, is to make a copy of the array with the astype() method.

```
import numpy as np
```

```
arr = np.array([1.1, 2.1, 3.1])
newarr = arr.astype(int)
print(newarr)
print(newarr.dtype)
```

```
[1 2 3]
int32
```

Check Number of Dimensions :

NumPy Arrays provides the ndim attribute that returns an integer that tells us how many dimensions the array have.

Check how many dimensions the arrays have:

```
import numpy as np

a = np.array(42)
b = np.array([1, 2, 3, 4, 5])
c = np.array([[1, 2, 3], [4, 5, 6]])
d = np.array([[[1, 2, 3], [4, 5, 6]], [[1, 2, 3], [4, 5, 6]]])

print(a.ndim)
print(b.ndim)
print(c.ndim)
print(d.ndim)
```

```
0
1
2
3
```

```
import numpy as np
# 5D-array
ar5 = np.array([1,2,3,4,5], ndmin = 5)
print(ar5)
print(ar5.ndim)
```

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

note : when you print dimension of array then use 'ndim' attribute but when you create dimension of array use 'ndmin'

- ***Minimum Dimensional Arrays :*** An array can have any number of dimensions. When the array is created, you can define the number of dimensions by using the ndmin argument.

Create an array with minimum dimensions and verify that it has 2 dimensions:

```
import numpy as np

arr = np.array([1, 2, 3,4,5], ndmin = 2)

print(arr)
print(arr.ndim)
```

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

- ***Higher Dimensional Arrays :*** An array can have any number of dimensions. When the array is created, you can define the number of dimensions by using the ndmin argument.

Create an array with 5 dimensions and verify that it has 5 dimensions:

```
import numpy as np

arr = np.array([1, 2, 3, 4], ndmin=5)

print(arr)
print('number of dimensions :', arr.ndim)
```

```
[[[[[1 2 3 4]]]]]
number of dimensions : 5
```

way to create array :

there are 5 ways to create an array:

1. using data structure i. e. list or tuple
2. instrinsic numpy array creation object(arange,ones,zeros,etc)
3. reading array from disk from some custom/std formate
4. using array from raw bytes through use of string or buffer
5. using special library functions i.e. random

- NumPy is used to work with arrays. The array object in NumPy is called ndarray.We can create a NumPy ndarray object by using the array() function

Create a NumPy ndarray Object

```
import numpy as np
a1 = np.array([12,13,14,15])
print(a1)
print(type(a1))
```

```
[12 13 14 15]
<class 'numpy.ndarray'>
```

- we can pass a list, tuple or any array-like object into the array() method

Use a tuple to create a NumPy array

```
import numpy as np

a2 = np.array((1, 2, 3, 4, 5))
print(a2)
print(type(a2))
```

```
[1 2 3 4 5]
<class 'numpy.ndarray'>
```

create array with zeros

- numpy.zeros(number of rows, number of columns) : return an array of zero elements

```
import numpy as np

ar = np.zeros((3,5))

print(ar)
print(ar.dtype)
print(ar.shape)
```

```
[[0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]]
float64
(3, 5)
```

```
import numpy as np

ar = np.zeros((1,1))

print(ar)
print(ar.dtype)
print(ar.shape)
```

```
[[0.]]
float64
(1, 1)
```

```
import numpy as np

ar = np.zeros((4,))

print(ar)
print(ar.dtype)
print(ar.shape)
```

```
[0. 0. 0. 0.]
float64
(4,)
```

create array with ones

- numpy.ones(number of rows, number of columns) : return an array of elements value 1.

```
import numpy as np

ar = np.ones((3,5))

print(ar)
print(ar.dtype)
print(ar.shape)
```

```
[[1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1.]]
float64
(3, 5)
```

numpy.arange(m, n) : it works same as range(m,n) of python and print the elements from the size of 'n-1' from m or in defalut value of 'm' is zero. it works similar to range() function.

```
import numpy as np

ar = np.arange(10)
print(ar)
```

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

```
import numpy as np

ar = np.arange(3, 8)
print(ar)
```

```
[3 4 5 6 7]
```

numpy.linspace(x,y,z) - it'll print 'z' number of element with same interval from 'x' to 'y-1' x : starting index || y : ending index || z : number of element

- generate 6 number between 1 to 5

```
import numpy as np

ar = np.linspace(1, 5, 6)
print(ar)
```

```
[1.  1.8 2.6 3.4 4.2 5. ]
```

- generate 6 number between 0 to 1

```
import numpy as np

x = np.linspace(0,1,6)
print(x)
```

```
[0.  0.2 0.4 0.6 0.8 1. ]
```

numpy.empty((i,j)) - generate an array with random number with 'i' number of rowa and 'j' number of columns.

- generate array of random numbers with 3 rows and 4 columns

```
import numpy as np

x = np.empty((3,4))
print(x)
```

```
[[1.29559052e-311 3.16202013e-322 0.00000000e+000 0.00000000e+000]
 [1.20953760e-312 3.12015643e-033 7.40331411e-038 6.52421341e-038]
 [2.21533620e+160 1.39804330e-076 7.86958905e-071 5.53615832e+174]]
```

- generate array of random numbers with 1 rows and 5 columns

```
import numpy as np

x = np.empty(5)
print(x)
```

```
[0.2 0.4 0.6 0.8 1. ]
```

b = numpy.empty_like((p)) - it'll return same structured/size array as already exist array named 'p' (where 'b' is an array one of them we created before.)

```
import numpy as np

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

```
[[1 2]
 [3 4]]
```

```
import numpy as np

b = np.empty_like((p))
print(b)
```

```
[[0 0]
 [0 0]]
```

numpy.identity(n) - return metrix with size of " n*n ".

```
import numpy as np

ar = np.identity(3)
print(ar)
```

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

array_name.reshape(i,j) - it'll resize and return array with size of (i*j) (number of rows(i),numbers of columns(j)).

```
import numpy as np

ar = np.arange((30))
arr = ar.reshape((3,10))

print("array : ",ar)
print("shaped array :\n", arr)
```

```
array :  [ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19
20 21 22 23
 24 25 26 27 28 29]
shaped array :
 [[ 0  1  2  3  4  5  6  7  8  9]
 [10 11 12 13 14 15 16 17 18 19]
 [20 21 22 23 24 25 26 27 28 29]]
```

array_name.ravel() - it'll remove all executed operations performed in any array and return you 1D-array

```
import numpy as np

ar = np.array([[1,2,3],[4,5,6]])
arr = ar.ravel()
print(ar)
print(arr)
```

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

Numpy Axes :

NUMPY AXES ARE LIKE AXES IN A COORDINATE SYSTEM

cordinatte system have axes : representation of axes with a value

Axes in metrix : all nD array do work with axis 0 and 1 but 1D array has only axis 1

NOTE : "axis-0" means "y-axis" or column and "axis-1" means "x-axis" or rows

axis and their meaning according to metrix :

columns : axis = 0

rows : axis = 1

- ***Let's see them in coding :***

```
import numpy as np

ar = np.array([[1,2,3],[4,5,6],[7,1,0]])

print("metrix :")
print(ar)

print("sum all elements :")
print(ar.sum())

print("sum of axis 0 :")
print(ar.sum(axis = 0))

print("sum of axis 1 :")
print(ar.sum(axis = 1))
```

```
metrix :
[[1 2 3]
 [4 5 6]
 [7 1 0]]
sum all elements :
29
sum of axis 0 :
[12  8  9]
sum of axis 1 :
[ 6 15  8]
```

Explaination :

- sum of axis 0 : [12 8 9] sum of [(1+4+7) (2+5+1) (3+6+0)]
- sum of axis 1 : [6 15 8] sum of [(1+2+3) (4+5+6) (7+1+0)]

Transpose of metrix : when we convert row into columns and columns to rows and get the new metrix is known as Transpose of metrix.

syntax : array_name.T

note: later 'T' must be in upper later

```
import numpy as np

ar = np.array([[1,2,3],[4,5,6],[7,8,9]])
print("metrix :")
```

```
print(ar)
print("transpose of metrix :")
at = ar.T
print(at)
```

```
metrix :
[[1 2 3]
 [4 5 6]
 [7 8 9]]
transpose of metrix :
[[1 4 7]
 [2 5 8]
 [3 6 9]]
```

- transpose of transpose metrix is real metrix

```
at.T
```

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

flat in array : This method is used as a 1_D iterator over N-dimensional arrays or we can say that The numpy.ndarray.flat() function is used as a 1_D iterator over N-dimensional arrays. It is not a subclass of, Python's built-in iterator object, otherwise it a numpy.flatiter instance.

SYNTAX : array_name.flat

NOTE - its method used as object

- flatiter is just the type of the iterator object returned by flat (docs). So all you need to know about it is that it's an iterator like any other.
- Obviously, flatten consumes more memory and cpu, as it creates a new array, while flat only creates the iterator object, which is super fast.
- If all you need is to iterate over the array, in a flat manner, use flat.
- If you need an actual flat array (for purposes other than just explicitly iterating over it), use flatten.
- it can convert all values into given value

```
import numpy as np

ar = np.arange(1,9).reshape(2,4);
print(ar)

ar.flat = 1;
print(ar)
```

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

- change the value in a range

```
import numpy as np

ar = np.arange(1,16).reshape(3,5);
print(ar)

ar.flat[3:6] = 0
ar.flat[8:10] = 9
print("Changing values in a range : \n", ar)
```

```
[[ 1  2  3  4  5]
 [ 6  7  8  9 10]
 [11 12 13 14 15]]
Changing values in a range :
 [[ 1  2  3  0  0]
 [ 0  7  8  9  9]
 [11 12 13 14 15]]
```

- you can use it as print indexed values

```
import numpy as np
x = np.arange(1, 7).reshape(2, 3)
print(x)

print("array[3] indexd element : ",x.flat[3])
```

```
[[1 2 3]
 [4 5 6]]
array[3] indexd element :  4
```

- array_name.argmax(): return index of element whose value is maximum

```
import numpy as np

one = np.array([12,13,1,2,4,6,99])

print("array : ",one)
print("index of max value in array : ",one.argmax())
```

```
array :  [12 13  1  2  4  6 99]
index of max value in array :  6
```

```
import numpy as np

two = np.array([[12,24],[24,56]])

print("array : \n",two)
print("index of max value in array : ",two.argmax())
```

```
array :
 [[12 24]
```

```
 [24 56]]
index of max value in array :  3
```

- with axis

```
import numpy as np

two = np.array([[12,24],[24,6]])

print("array : \n",two)
print("index of max value in array : ",two.argmax(axis = 0))
```

```
array :
 [[12 24]
 [24  6]]
index of max value in array :  [1 0]
```

```
import numpy as np

two = np.array([[12,4],[24,56]])

print("array : \n",two)
print("index of max value in array : ",two.argmax(axis = 1))
```

```
array :
 [[12  4]
 [24 56]]
index of max value in array :  [0 1]
```

- array_name.argmin(): return index of element whose value is minimum

```
import numpy as np

one = np.array([12,13,1,2,4,6,99])

print("array : ",one)
print("index of max value in array : ",one.argmin())
```

```
array :  [12 13  1  2  4  6 99]
index of max value in array :  2
```

```
import numpy as np

two = np.array([[12,24],[24,56]])

print("array : \n",two)
print("index of max value in array : ",two.argmin())
```

```
array :
 [[12 24]
 [24 56]]
index of max value in array :  0
```

- with axis

```
import numpy as np

two = np.array([[12,24],[24,6]])

print("array : \n",two)
print("index of max value in array : ",two.argmax(axis = 0))
```

```
array :
 [[12 24]
 [24  6]]
index of max value in array :  [1 0]
```

```
import numpy as np

two = np.array([[12,4],[24,56]])

print("array : \n",two)
print("index of max value in array : ",two.argmax(axis = 1))
```

```
array :
 [[12  4]
 [24 56]]
index of max value in array :  [0 1]
```

- array_name.argsort() : return index of element of u put element of that index then your index will be sorted

```
import numpy as np

ar = np.array([[1, 2, 3],[4, 5, 6],[7, 1, 0]])
print("array : \n",ar)

print("indexes of array in sorted form :\n",ar.argsort())
```

```
array :
 [[1 2 3]
 [4 5 6]
 [7 1 0]]
indexes of array in sorted form :
 [[0 1 2]
 [0 1 2]
 [2 1 0]]
```

- with axis

```
import numpy as np

ar = np.array([[1, 2, 3],[4, 5, 6],[7, 1, 0]])
print("array : \n",ar)
```

```
print("indexes of array in sorted form :\n",ar.argsort(axis = 0))
```

```
array :
 [[1 2 3]
 [4 5 6]
 [7 1 0]]
indexes of array in sorted form :
 [[0 2 2]
 [1 0 0]
 [2 1 1]]
```

```
import numpy as np

ar = np.array([[1, 2, 3],[4, 5, 6],[7, 1, 0]])
print("array : \n",ar)

print("indexes of array in sorted form :\n",ar.argsort(axis=1))
```

```
array :
 [[1 2 3]
 [4 5 6]
 [7 1 0]]
indexes of array in sorted form :
 [[0 1 2]
 [0 1 2]
 [2 1 0]]
```

basic math in numpy

- let's create 2 array for see math operations:

```
import numpy as np

ar1 = np.array([[1,2,3],[4,5,6],[7,1,0]])
ar2 = np.array([[1,2,1],[4,0,6],[8,1,0]])

print(f"array 1 : \n{ar1} \narray 2 : \n{ar2}")
```

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

- sum of array

```
ar = ar1 + ar2
print(ar)
```

```
[[ 2  4  4]
 [ 8  5 12]
 [15  2  0]]
```

- sub of array

```
ar = ar1 - ar2
print(ar)
```

```
[[ 0  0  2]
 [ 0  5  0]
 [-1  0  0]]
```

```
ar = ar2 - ar1
print(ar)
```

```
[[ 0  0 -2]
 [ 0 -5  0]
 [ 1  0  0]]
```

- ltiply of array

```
ar = ar1 * ar2
print(ar)
```

```
[[ 1  4  3]
 [16  0 36]
 [56  1  0]]
```

```
ar = 2 * ar1
print(ar)
```

```
[[ 2  4  6]
 [ 8 10 12]
 [14  2  0]]
```

- division of array

```
ar = ar1 / ar2
print(ar)
```

```
[[1.    1.    3.   ]
 [1.      inf 1.   ]
 [0.875 1.      nan]]

C:\Users\rajen\AppData\Local\Temp\ipykernel_1828\3445177483.py:1:
RuntimeWarning: divide by zero encountered in divide
  ar = ar1 / ar2
C:\Users\rajen\AppData\Local\Temp\ipykernel_1828\3445177483.py:1:
RuntimeWarning: invalid value encountered in divide
  ar = ar1 / ar2
```

```
ar = ar2 / ar1
print(ar)
```

```
[[1.         1.         0.33333333]
 [1.         0.         1.        ]
 [1.14285714 1.                nan]]

C:\Users\rajen\AppData\Local\Temp\ipykernel_1828\557195517.py:1:
RuntimeWarning: invalid value encountered in divide
  ar = ar2 / ar1
```

- maximum value of array :

```
import numpy as np

ar = np.array([[1,2,1],[4,0,6],[8,1,0]])

print(f"array :\n{ar}")
print(f"max value is : {ar.max()}")
```

```
array :
[[1 2 1]
 [4 0 6]
 [8 1 0]]
max value is : 8
```

- minimum value of array :

```
import numpy as np

ar = np.array([[1,2,1],[4,0,6],[8,1,0]])

print(f"array :\n{ar}")
print(f"max value is : {ar.min()}")
```

```
array :
[[1 2 1]
 [4 0 6]
 [8 1 0]]
max value is : 0
```

- sort of array :

```
import numpy as np

ar = np.array([10,13,12,9,4,7])

print(f"array :\n{ar}")
print(f"sort array is : {np.sort(ar)}")
```

```
array :
[10 13 12  9  4  7]
sort array is : [ 4  7  9 10 12 13]
```

```
import numpy as np

ar = np.array([[1,2,1],[4,0,6],[8,1,0]])

print(f"array :\n{ar}")
print(f"sort array is : {np.sort(ar)}")
```

```
array :
[[1 2 1]
 [4 0 6]
 [8 1 0]]
sort array is : [[1 1 2]
 [0 4 6]
 [0 1 8]]
```

- total number of non-zero elements in array :

```
import numpy as np

ar3 = np.array([[11,12,13],
                [0,14,0],
                [21,0,14]])
print(ar3)
# count non-zero elements
print("total number of non-zero elements in
array :",np.count_nonzero(ar3))
```

```
[[11 12 13]
 [ 0 14  0]
 [21  0 14]]
total number of non-zero elements in array : 6
```

- find index of elements with conditions

```
import numpy as np
ar1 = np.array([[1,2,3],[4,5,6],[7,1,0]])

# find index of elements with conditions
max_than_4 = np.where(ar1>4)
print(max_than_4)

# hint: array(axis 0->[1,1,2] axis 1->[1,2,0])
print(type(max_than_4))
```

```
(array([1, 1, 2], dtype=int64), array([1, 2, 0], dtype=int64))
<class 'tuple'>
```

```
import numpy as np

ar1 = np.array([[1,2,3],[4,5,6],[7,1,0]])

max_than_4 = np.argwhere(ar1>4)
print(max_than_4)

# hint: array(axis 0->[1,1,2] axis 1->[1,2,0])
print(type(max_than_4))
```

```
[[1 1]
 [1 2]
 [2 0]]
<class 'numpy.ndarray'>
```

WAP to show that numpy module consume less memories then normal coding

```
import sys
import numpy as np

py_ar = [0,4,55,2]
np_ar = np.array([0,4,55,2])

# get total size  of it using  sys.getsizeof(args) multiply by number
of items
x = sys.getsizeof(1) * len(py_ar)
print(x)

# get total size  of it using  array.itemsize(args) multiply by number
of items
y = np_ar.itemsize * np_ar.size
print(y)

# get total size  of it using  array.itemsize(args) multiply by number
of items
y = np_ar.itemsize * len(np_ar)
print(y)
```

```
112
16
16
```

Quick revise some other numpy features:

indexing in array:

- ***positive index:***
- 1D array elements using index/ traditional method :

```
import numpy as np
```

```
ar1 = np.array([1,2,3,4,5])
print(ar1)
print(ar1.ndim)
print('first element = ',ar1[0])
print('third element = ',ar1[2])
```

```
[1 2 3 4 5]
1
first element =  1
third element =  3
```

- 2D array elements using index/ traditional method :

```
import numpy as np

ar2 = np.array([[1,2,3,4,5],[6,7,8,9,10]])

print(ar2)
print("dimention of array : ",ar2.ndim)

print('3rd row 2nd element : ',ar2[1,2])  # array_name[col,row]
print('2nd row 5th element : ',ar2[1,4])  # Access the element on the
2nd row, 5th column
```

```
[[ 1  2  3  4  5]
 [ 6  7  8  9 10]]
dimention of array :  2
3rd row 2nd element :  8
2nd row 5th element :  10
```

- 3D array elements using index/ traditional method :

```
import numpy as np

ar3 = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])

print(ar3)
print(' third element of the second array of the first array',ar3[0,
1, 2])
```

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

 [[ 7  8  9]
  [10 11 12]]]
 third element of the second array of the first array 6
```

]]]

Example Explained arr[0, 1, 2] prints the value 6. And this is why: The first number represents the first dimension, which contains two arrays: [[1, 2, 3], [4, 5, 6]] and: [[7, 8, 9], [10, 11, 12]] Since

we selected 0, we are left with the first array: [[1, 2, 3], [4, 5, 6]] The second number represents the second dimension, which also contains two arrays: [1, 2, 3] and: [4, 5, 6] Since we selected 1, we are left with the second array: [4, 5, 6] The third number represents the third dimension, which contains three values: 4 5 6 Since we selected 2, we end up with the third value: 6

- ***Negative index :***

```
import numpy as np

arr = np.array([[1,2,3,4,5], [6,7,8,9,10]])

print(arr)
print('Last element from 2nd dim: ', arr[1, -1])
```

```
[[ 1  2  3  4  5]
 [ 6  7  8  9 10]]
Last element from 2nd dim:  10
```

Slicing arrays-

Slicing in python means taking elements from one given index to another given index. We pass slice instead of index like this: [start:end]. We can also define the step, like this: [start:end:step]. If we don't pass start its considered 0 If we don't pass end its considered length of array in that dimension If we don't pass step its considered 1

example:

```
import numpy as np

arr = np.array([0, 1, 2, 3, 4, 5, 6, 7])
print('array = ',arr)
print('array[1:5]',arr[1:5])
print('array[:5]',arr[:5])
print('array[4:]',arr[4:])
print('array[1:6:2]',arr[1:6:2])
print('array[-3:-1]',arr[-3:-1])
```

```
array =  [0 1 2 3 4 5 6 7]
array[1:5] [1 2 3 4]
array[:5] [0 1 2 3 4]
array[4:] [4 5 6 7]
array[1:6:2] [1 3 5]
array[-3:-1] [5 6]
```

NOTE: when you leave empty of any index in this method that means 0 from start/end.

note : slicing or split array using function see later

NumPy Array Copy vs View:

The Difference Between Copy and View - 1. The main difference between a copy and a view of an array is that the copy is a new array, and the view is just a view of the original array. 2. The copy owns the data and any changes made to the copy will not affect original array, and any changes made to the original array will not affect the copy. 3. The view does not own the data and any changes made to the view will affect the original array, and any changes made to the original array will affect the view.

- COPY : Make a new array as copy, change the values in array, and display both arrays.

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5])
x = arr.copy()

# change value in original array
arr[0] = 42
print(arr)
print(x)
```

```
[42  2  3  4  5]
[1 2 3 4 5]
```

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5])
x = arr.copy()

#  make change in copy
x[0] = 42
print(arr)
print(x)
```

```
[1 2 3 4 5]
[42  2  3  4  5]
```

- VIEW : Make a view, change the values array, and display both arrays:

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5])
x = arr.view()

# change value in original array
arr[0] = 42
print(arr)
print(x)
```

```
[42  2  3  4  5]
[42  2  3  4  5]
```

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5])
x = arr.view()

#  make change in copy
x[0] = 42
print(arr)
print(x)
```

```
[42  2  3  4  5]
[42  2  3  4  5]
```

Check if Array Owns its Data

As mentioned above, copies owns the data, and views does not own the data, but how can we check this? Every NumPy array has the attribute base that returns None if the array owns the data. Otherwise, the base attribute refers to the original object.

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5])

x = arr.copy()
y = arr.view()

print(x.base)
print(y.base)
```

```
None
[1 2 3 4 5]
```

let's revise shape() and reshape() function ***What does the shape tuple represent?*** Integers at every index tells about the number of elements the corresponding dimension has In the example above at index-4 we have value 4, so we can say that 5th (4 + 1 th) dimension has 4 elements.

Example :

- Print the shape of a 2-D array:

```
import numpy as np

arr = np.array([[1, 2, 3, 4], [5, 6, 7, 8]])
print(arr.shape)
```

```
(2, 4)
```

- Reshape From 1-D to 2-D array:

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12])
newarr = arr.reshape(4, 3)
print(newarr)
```

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

- Reshape From 1-D to 2-D:

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12])
newarr = arr.reshape(2, 3, 2)
# The outermost dimension will have 2 arrays that contains 3 arrays,
each with 2 elements:
print(newarr)
```

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

 [[ 7  8]
  [ 9 10]
  [11 12]]]
```

Unknown Dimension : You are allowed to have one "unknown" dimension. Meaning that you do not have to specify an exact number for one of the dimensions in the reshape method. Pass -1 as the value, and NumPy will calculate this number for you.

- Convert 1D array with 8 elements to 3D array with 2x2 elements:

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5, 6, 7, 8])
newarr = arr.reshape(2, 2, -1)

print(newarr)
```

```
[[[1 2]
  [3 4]]

 [[5 6]
  [7 8]]]
```

Flattening the arrays : Flattening array means converting a multidimensional array into a 1D array. We can use reshape(-1) to do this.

```
import numpy as np

arr = np.array([[1, 2, 3], [4, 5, 6]])
newarr = arr.reshape(-1)

print(newarr)
```

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

Note : There are a lot of functions for changing the shapes of arrays in numpy flatten, ravel and also for rearranging the elements rot90, flip, fliplr, flipud etc. These fall under Intermediate to Advanced section of numpy.

Iterating Arrays

Iterating means going through elements one by one. As we deal with multi-dimensional arrays in numpy, we can do this using basic for loop of python. If we iterate on a 1-D array it will go through each element one by one.

- Iterate on the elements of the following 1-D array:

```
import numpy as np

arr = np.array([1, 2, 3])

for x in arr:
    print(x)
```

```
1
2
3
```

- Iterate on the elements of the following 2-D array:

```
import numpy as np

arr = np.array([[1, 2, 3], [4, 5, 6]])

for x in arr:
    print(x)
```

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

- Iterate on the elements of the following 3-D array:

```
import numpy as np

arr = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])
for x in arr:
    print(x)
```

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

more : If we iterate on a n-D array it will go through n-1th dimension one by one.

- To return the actual values, the scalars, we have to iterate the arrays in each dimension.
- Iterate on each scalar element of the 2-D array:

```
import numpy as np

arr = np.array([[1, 2, 3], [4, 5, 6]])

for x in arr:
    for y in x:
        print(y)
```

```
1
2
3
4
5
6
```

- n a 3-D array it will go through all the 2-D arrays.
- Iterate on the elements of the following 3-D array:

```
import numpy as np

arr = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])
for x in arr:
    print(x)
```

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

- To return the actual values, the scalars, we have to iterate the arrays in each dimension.
- Iterate down to the scalars:

```
import numpy as np

arr = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])

for x in arr:
    for y in x:
        for z in y:
            print(z)
```

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

Iterating Arrays Using nditer()

The function nditer() is a helping function that can be used from very basic to very advanced iterations. It solves some basic issues which we face in iteration, lets go through it with examples.

Iterating on Each Scalar Element - In basic for loops, iterating through each scalar of an array we need to use n for loops which can be difficult to write for arrays with very high dimensionality.

- Iterate through the following 2-D array:

```
import numpy as np

arr = np.array([[0,2,1],[3,4,5]])
for x in np.nditer(arr):
    print(x)
```

```
0
2
1
3
4
5
```

- Iterate through the following 3-D array:

```
import numpy as np

arr = np.array([[[1, 2], [3, 4]], [[5, 6], [7, 8]]])
for x in np.nditer(arr):
    print(x)
```

```
1
2
3
4
5
```

```
6
7
8
```

- Iterating Array With Different Data Types

  We can use op_dtypes argument and pass it the expected datatype to change the datatype of elements while iterating. NumPy does not change the data type of the element in-place (where the element is in array) so it needs some other space to perform this action, that extra space is called buffer, and in order to enable it in nditer() we pass flags=['buffered'].
- Iterate through the array as a string:

```
import numpy as np

arr = np.array([1, 2, 3])
for x in np.nditer(arr, flags=['buffered'], op_dtypes=['S']):
    print(x)

b'1'
b'2'
b'3'
```

- Iterating With Different Step Size

  We can use filtering and followed by iteration. Iterate through every scalar element of the 2D array skipping 1 element:

```
import numpy as np

arr = np.array([[1, 2, 3, 4], [5, 6, 7, 8]])
for x in np.nditer(arr[:, ::2]):
    print(x)

1
3
5
7

import numpy as np

arr = np.array([[0, 1, 2, 3, 4], [5, 6, 7, 8, 9]])
for x in np.nditer(arr[:, ::2]):
    print(x)

0
2
4
5
7
9
```

- Enumerated Iteration Using ndenumerate()
  Enumeration means mentioning sequence number of somethings one by one. Sometimes we require corresponding index of the element while iterating, the ndenumerate() method can be used for those usecases.
- Enumerate on following 1D arrays elements:

```
import numpy as np

arr = np.array([1, 2, 3])

for idx, x in np.ndenumerate(arr):
    print(idx, x)
```

```
(0,) 1
(1,) 2
(2,) 3
```

- Enumerate on following 2D arrays elements:

```
import numpy as np

arr = np.array([[1, 2, 3, 4], [5, 6, 7, 8]])

print("array : \n", arr,"\n")

for idx, x in np.ndenumerate(arr):
    print(idx, x)
```

```
array :
 [[1 2 3 4]
 [5 6 7 8]]

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

```
import numpy as np

arr = np.array([[1, 2, 3, 4], [5, 6, 7, 8],[11, 12, 13, 14]])

print("array : \n", arr,"\n")

for idx, x in np.ndenumerate(arr):
    print(idx, x)
```

```
array :
 [[ 1  2  3  4]
```

```
 [ 5  6  7  8]
 [11 12 13 14]]

(0, 0) 1
(0, 1) 2
(0, 2) 3
(0, 3) 4
(1, 0) 5
(1, 1) 6
(1, 2) 7
(1, 3) 8
(2, 0) 11
(2, 1) 12
(2, 2) 13
(2, 3) 14
```

Join numpy Arrays : 1. concate Functions

Joining means putting contents of two or more arrays in a single array. In SQL we join tables based on a key, whereas in NumPy we join arrays by axes. We pass a sequence of arrays that we want to join to the concatenate() function, along with the axis. If axis is not explicitly passed, it is taken as 0.

syntax : numpy.concatenate((array_name1,array_name2)) where, concate/join array_name1 and array_name2

- Join two 1D arrays :

```
import numpy as np
ar1 = np.array([[1,2,3]])
ar2 = np.array([[4,5,6]])
ar = np.concatenate((ar1,ar2))
print(ar)

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

- Join two 2D arrays :

```
import numpy as np
ar1 = np.array([[1,2,3],[3,4,5]])
ar2 = np.array([[5,6,7],[7,8,9]])
ar = np.concatenate((ar1,ar2))  # by default axis = 0
print(ar)

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

- using Axis concate arrays

- using Axis concate arrays for axis = 0

```
import numpy as np
ar1 = np.array([[1,2,3],[3,4,5]])
ar2 = np.array([[5,6,7],[7,8,9]])

ar = np.concatenate((ar1,ar2), axis = 1) # add in rows
print(ar)
```

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

- using Axis concate arrays for axis = 1

```
import numpy as np
ar1 = np.array([[1,2,3],[3,4,5]])
ar2 = np.array([[5,6,7],[7,8,9]])

ar = np.concatenate((ar1,ar2), axis = 0) # add in cols
print(ar)
```

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

Join numpy Arrays : 2. Stack Functions

- Stacking is same as concatenation, the only difference is that stacking is done along a new axis.
- We can concatenate two 1-D arrays along the second axis which would result in putting them one over the other, ie. stacking.
- We pass a sequence of arrays that we want to join to the stack() method along with the axis. If axis is not explicitly passed it is taken as 0.
- using axis :

```
import numpy as np

arr1 = np.array([1, 2, 3])
arr2 = np.array([4, 5, 6])

arr = np.stack((arr1, arr2), axis=1)

print(arr)
```

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

```
import numpy as np

arr1 = np.array([1, 2, 3])
arr2 = np.array([4, 5, 6])

arr = np.stack((arr1, arr2), axis=0)

print(arr)
```

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

- Stacking Along Rows : ***hstack((a,b,c...))*** - NumPy provides a helper function: hstack() to stack along rows. where a,b,c... are 1D arrays

horizontally joint

```
import numpy as np

arr1 = np.array([1, 2, 3])
arr2 = np.array([4, 5, 6])

arr = np.hstack((arr1, arr2))

print(arr)
```

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

- Stacking Along Columns : ***vstack((a,b,c...))*** - NumPy provides a helper function: vstack() to stack along columns. where a,b,c... are 1D arrays.

verticlally joint

```
import numpy as np

arr1 = np.array([1, 2, 3])
arr2 = np.array([4, 5, 6])

arr = np.vstack((arr1, arr2))

print(arr)
```

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

- Stacking Along Height (depth) : ***dstack((a,b,c...))*** - NumPy provides a helper function: dstack() to stack along height, which is the same as depth. where a,b,c... are arrays

```
import numpy as np

arr1 = np.array([1, 2, 3])
```

```
arr2 = np.array([4, 5, 6])

arr = np.dstack((arr1, arr2))

print(arr)
```

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

NumPy Splitting :

it is just reverse operations of join arrays, slipting means divide collection into piceas.

numpy.array() -Splitting is reverse operation of Joining. Joining merges multiple arrays into one and Splitting breaks one array into multiple. We use array_split() for splitting arrays, we pass it the array we want to split and the number of splits.

- split array into 2 parts:

```
import numpy as np

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

newarr = np.array_split(arr, 2)

print(newarr)
```

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

```
import numpy as np

arr = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12], [13,
14, 15], [16, 17, 18]])

newarr = np.array_split(arr, 2)

print(newarr)
```

```
[array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]]), array([[10, 11, 12],
       [13, 14, 15],
       [16, 17, 18]])]
```

- split array into 3 parts:

```
import numpy as np

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

newarr = np.array_split(arr, 3)
```

```
print(newarr)
```

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

```
import numpy as np

arr = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12], [13,
14, 15], [16, 17, 18]])

newarr = np.array_split(arr, 3)

print(newarr)
```

```
[array([[1, 2, 3],
       [4, 5, 6]]), array([[ 7,  8,  9],
       [10, 11, 12]]), array([[13, 14, 15],
       [16, 17, 18]])]
```

- split array into 4 parts:

```
import numpy as np

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

newarr = np.array_split(arr, 4)

print(newarr)
```

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

```
import numpy as np

arr = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12], [13,
14, 15], [16, 17, 18]])

newarr = np.array_split(arr, 4)

print(newarr)
```

```
[array([[1, 2, 3],
       [4, 5, 6]]), array([[ 7,  8,  9],
       [10, 11, 12]]), array([[13, 14, 15]]), array([[16, 17, 18]])]
```

- split array into 5 parts:

```
import numpy as np

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

newarr = np.array_split(arr, 5)
```

```
print(newarr)
```

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

```
import numpy as np

arr = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12], [13,
14, 15], [16, 17, 18]])

newarr = np.array_split(arr, 5)

print(newarr)
```

```
[array([[1, 2, 3],
       [4, 5, 6]]), array([[7, 8, 9]]), array([[10, 11, 12]]),
array([[13, 14, 15]]), array([[16, 17, 18]])]
```

- split array into 6 parts:

```
import numpy as np

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

newarr = np.array_split(arr, 6)

print(newarr)
```

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

Note: The return value is an array containing three arrays.

```
import numpy as np

arr = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12], [13,
14, 15], [16, 17, 18]])

newarr = np.array_split(arr, 6)

print(newarr)
```

```
[array([[1, 2, 3]]), array([[4, 5, 6]]), array([[7, 8, 9]]),
array([[10, 11, 12]]), array([[13, 14, 15]]), array([[16, 17, 18]])]
```

split the array with hstack(), vstack() and dstack() :

```
import numpy as np

arr = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12], [13,
```

```
14, 15], [16, 17, 18]])

newarr1 = np.hsplit(arr, 3)
newarr2 = np.vsplit(arr, 3)
# newarr3 = np.dsplit(arr, 3)   #dsplit only works on arrays of 3 or
more dimensions

print('hstack() : \n ',newarr1)
print('vstack() : \n ',newarr2)
# print('dstack() : \n ',newarr3)
```

```
hstack() :
  [array([[ 1],
        [ 4],
        [ 7],
        [10],
        [13],
        [16]]), array([[ 2],
        [ 5],
        [ 8],
        [11],
        [14],
        [17]]), array([[ 3],
        [ 6],
        [ 9],
        [12],
        [15],
        [18]])]
vstack() :
  [array([[1, 2, 3],
        [4, 5, 6]]), array([[ 7,  8,  9],
        [10, 11, 12]]), array([[13, 14, 15],
        [16, 17, 18]])]
```

NumPy Searching Arrays:

You can search an array for a certain value, and return the indexes that get a match. To search an array, use the where() method.

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5, 4, 4])
x = np.where(arr == 4)

print(x)
```

```
(array([3, 5, 6], dtype=int64),)
```

- Find the indexes where the values are even and odd

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5, 6, 7, 8])

x = np.where(arr%2 == 0)
y = np.where(arr%2 == 1)

print("even : ",x)
print("odd : ", y)
```

```
even :  (array([1, 3, 5, 7], dtype=int64),)
odd :  (array([0, 2, 4, 6], dtype=int64),)
```

- ***numpy.searchsorted(array, value)*** - There is a method called searchsorted() which ***performs a binary search in the array,*** and returns the ***index*** where the specified value would be inserted to maintain the search order.

```
import numpy as np

arr = np.array([6, 7, 8, 9, 3])
x = np.searchsorted(arr, 3)

print(x)
```

```
0
```

Search From the Right Side - for it we can give side='right' to return the right most index instead.

```
import numpy as np

arr = np.array([6, 7, 8, 9])

x = np.searchsorted(arr, 7, side='right')

print(x)
```

```
2
```

- To search for ***more than one value***, use an array with the specified values.
- Find the indexes where the values 2, 4, and 6 should be inserted:

```
import numpy as np

arr = np.array([1, 3, 5, 7])

x = np.searchsorted(arr, [2, 4, 6])

print(x)
```

```
[1 2 3]
```

Sorting Arrays:

numpy.sort() : "Sorting means putting elements in an ordered sequence" Ordered sequence is any sequence that has an order corresponding to elements, like numeric or alphabetical, ascending or descending.

```
import numpy as np

arr = np.array([3, 2, 0, 1])

print(np.sort(arr))
```

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

Note: This method returns a copy of the array, leaving the original array unchanged.

- Sort the array alphabetically:

```
import numpy as np

arr = np.array(['banana', 'cherry', 'apple'])

print(np.sort(arr))
```

```
['apple' 'banana' 'cherry']
```

- Sort a boolean array:

```
import numpy as np

arr = np.array([True, False, True])

print(np.sort(arr))
```

```
[False  True  True]
```

sort 2D array :

```
import numpy as np

arr = np.array([[3, 2, 4], [5, 0, 1]])

print(np.sort(arr))
```

```
[[2 3 4]
 [0 1 5]]
```

NumPy Filter Array:

Getting some elements out of an existing array and creating a new array out of them is called filtering. In NumPy, you filter an array using a boolean index list. ***A boolean index list is a list of***

booleans corresponding to indexes in the array If the value at an index is True that element is contained in the filtered array, if the value at that index is False that element is excluded from the filtered array.

- Create an array from the elements on index 0 and 2:

```
import numpy as np

arr = np.array([41, 42, 43, 44])
x = [True, False, True, False]

newarr = arr[x]
print(newarr)
```

```
[41 43]
```

note : but the common use is to create a filter array based on conditions.

- Create a filter array that will return only values higher than 42:

```
import numpy as np

arr = np.array([45, 41, 42, 43, 44])
# create empty list
filter_arr = []

# go through each element in arr
for element in arr:
# if the element is higher than 42, set the value to True, otherwise
False:
    if element > 42:
        filter_arr.append(True)
    else:
        filter_arr.append(False)

newarr = arr[filter_arr]

print(filter_arr)
print(newarr)
```

```
[True, False, False, True, True]
[45 43 44]
```

- Create a filter array that will return only even elements from the original array:

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5, 6, 7])

# Create an empty list
filter_arr = []
```

```
# go through each element in arr
for element in arr:
  # if the element is completely divisble by 2, set the value to True,
otherwise False
  if element % 2 == 0:
    filter_arr.append(True)
  else:
    filter_arr.append(False)

newarr = arr[filter_arr]

print(filter_arr)
print(newarr)
```

```
[False, True, False, True, False, True, False]
[2 4 6]
```

Random Numbers in NumPy :

Random number- It does NOT mean a different number every time. Random means something that can not be predicted logically.

- create random number of array :

```
import numpy as np

x = np.random.normal(170, 10, 250)
print(x)
```

```
[183.84771093 158.21761559 180.03778763 164.54916094 181.35944568
 162.98211371 146.03634357 166.5937731  182.97873518 146.92344377
 191.09140385 162.48511398 178.96815794 186.18714093 175.13485857
 173.26274813 161.36963508 166.71063241 171.87533235 172.86588916
 183.97849336 175.96710402 178.4032555  173.60237603 168.24301318
 164.97620835 168.40294392 163.36827569 176.78587578 179.27336358
 170.72708676 155.35476402 167.88843223 181.70718095 170.17656858
 179.46725208 169.08290999 167.9791582  175.5663438  176.9463989
 172.69179411 189.21766812 169.46321537 176.24221511 182.3145498
 159.35510592 168.3392226  172.72144903 184.08209821 164.36723482
 167.68009679 164.78904782 158.76902984 163.7011288  161.43974906
 170.12240698 166.91365571 163.28961679 168.1492044  157.69812369
 177.9346833  182.31859281 173.11972521 166.78804287 167.43803249
 173.2540722  172.45985784 167.19158265 174.44699392 192.09289403
 155.39645672 180.06440027 163.45755569 161.49799389 168.72954698
 152.37934793 165.55915713 204.19058408 169.51694563 152.54572833
 162.78608529 174.1053053  179.88437986 158.90496122 172.80996411
 168.88431649 171.90849662 187.88167469 175.09349767 175.48194946
 166.50487513 171.13540634 193.42260512 184.6255176  168.88383982
 184.75995485 181.46133967 157.17270752 172.23442842 166.08978578
 168.9967386  166.23241441 173.33455014 172.32363481 156.74621063
```

```
 162.2027058  177.10058253 161.96353037 152.59484549 171.27369236
 171.58418621 169.03224265 172.72791023 163.48445944 174.13182627
 152.66648497 177.07593355 183.88499561 188.13069411 166.26613556
 164.24097099 165.24865285 182.70782936 175.98607407 168.80688919
 166.59610786 173.53559143 155.82047039 178.55141939 184.09458298
 152.51707104 186.26160224 171.7444734  181.40739401 180.8988085
 183.91737235 151.38815889 183.31189855 160.83004237 158.63888187
 162.14465331 182.33218387 148.57680803 198.61678101 165.68312759
 163.74381562 176.55979094 171.29203033 154.42198338 178.33838485
 158.33100391 177.54049346 170.78855442 175.4959242  158.32950905
 172.6287802  158.24061853 170.47136453 159.25594048 169.88577936
 148.41757302 156.50891804 153.82538328 172.62864624 166.8961667
 171.27656716 192.69919609 185.18893387 173.29545591 186.82588902
 165.09543666 183.63422093 173.64743985 173.18990086 166.47139704
 164.19037364 182.64170557 170.83455259 164.63751942 175.52938805
 166.68892492 160.75824064 183.81662729 167.25112623 172.1682717
 173.79038725 172.87078582 163.03927106 162.13411823 171.25541653
 166.69220186 166.1602112  153.33958203 165.15768048 170.01498063
 174.49307314 179.23333574 163.17064143 165.04525936 167.84965261
 154.49982094 187.92287647 175.20737071 177.20319163 164.23612171
 167.30046056 171.61273451 163.75141277 164.51867122 162.14165877
 181.13511798 175.57570098 172.74458797 164.97068681 179.70612742
 192.70470353 184.88092127 154.34090521 171.11504196 164.11659864
 181.02942268 175.35615389 179.32498131 165.88181734 167.90296967
 168.91728379 177.35713167 176.21330784 182.15424346 166.84965718
 176.35774265 181.14437668 150.20053814 184.38737888 174.30151339
 158.1437301  190.61858337 173.89107147 183.41464175 172.57887054
 166.05881063 149.63389801 177.63470545 147.76409945 162.66591321
 166.69019174 172.74477483 186.23276788 161.51639343 165.87434341]
```

NOTE : It is not important topic in data science if u want to study u can go for official documentation and w3schoool

Seaborn :

Visualize Distributions With Seaborn Seaborn is a library that uses Matplotlib underneath to plot graphs. It will be used to visualize random distributions.

```
pip install seaborn
```

```
Collecting seaborn
  Using cached seaborn-0.13.2-py3-none-any.whl.metadata (5.4 kB)
Requirement already satisfied: numpy!=1.24.0,>=1.20 in c:\users\rajen\
anaconda3\envs\notebook\lib\site-packages (from seaborn) (1.26.4)
Requirement already satisfied: pandas>=1.2 in c:\users\rajen\
anaconda3\envs\notebook\lib\site-packages (from seaborn) (2.2.0)
Requirement already satisfied: matplotlib!=3.6.1,>=3.4 in c:\users\
```

```
rajen\anaconda3\envs\notebook\lib\site-packages (from seaborn) (3.8.3)
Requirement already satisfied: contourpy>=1.0.1 in c:\users\rajen\
anaconda3\envs\notebook\lib\site-packages (from matplotlib!
=3.6.1,>=3.4->seaborn) (1.2.0)
Requirement already satisfied: cycler>=0.10 in c:\users\rajen\
anaconda3\envs\notebook\lib\site-packages (from matplotlib!
=3.6.1,>=3.4->seaborn) (0.12.1)
Requirement already satisfied: fonttools>=4.22.0 in c:\users\rajen\
anaconda3\envs\notebook\lib\site-packages (from matplotlib!
=3.6.1,>=3.4->seaborn) (4.49.0)
Requirement already satisfied: kiwisolver>=1.3.1 in c:\users\rajen\
anaconda3\envs\notebook\lib\site-packages (from matplotlib!
=3.6.1,>=3.4->seaborn) (1.4.5)
Requirement already satisfied: packaging>=20.0 in c:\users\rajen\
anaconda3\envs\notebook\lib\site-packages (from matplotlib!
=3.6.1,>=3.4->seaborn) (23.1)
Requirement already satisfied: pillow>=8 in c:\users\rajen\anaconda3\
envs\notebook\lib\site-packages (from matplotlib!=3.6.1,>=3.4-
>seaborn) (10.2.0)
Requirement already satisfied: pyparsing>=2.3.1 in c:\users\rajen\
anaconda3\envs\notebook\lib\site-packages (from matplotlib!
=3.6.1,>=3.4->seaborn) (3.1.1)
Requirement already satisfied: python-dateutil>=2.7 in c:\users\rajen\
anaconda3\envs\notebook\lib\site-packages (from matplotlib!
=3.6.1,>=3.4->seaborn) (2.8.2)
Requirement already satisfied: pytz>=2020.1 in c:\users\rajen\
anaconda3\envs\notebook\lib\site-packages (from pandas>=1.2->seaborn)
(2023.3.post1)
Requirement already satisfied: tzdata>=2022.7 in c:\users\rajen\
anaconda3\envs\notebook\lib\site-packages (from pandas>=1.2->seaborn)
(2024.1)
Requirement already satisfied: six>=1.5 in c:\users\rajen\anaconda3\
envs\notebook\lib\site-packages (from python-dateutil>=2.7-
>matplotlib!=3.6.1,>=3.4->seaborn) (1.16.0)
Using cached seaborn-0.13.2-py3-none-any.whl (294 kB)
Installing collected packages: seaborn
Successfully installed seaborn-0.13.2
Note: you may need to restart the kernel to use updated packages.

pip install matplotlib

Requirement already satisfied: matplotlib in c:\users\rajen\anaconda3\
envs\notebook\lib\site-packages (3.8.3)
Requirement already satisfied: contourpy>=1.0.1 in c:\users\rajen\
anaconda3\envs\notebook\lib\site-packages (from matplotlib) (1.2.0)
Requirement already satisfied: cycler>=0.10 in c:\users\rajen\
anaconda3\envs\notebook\lib\site-packages (from matplotlib) (0.12.1)
Requirement already satisfied: fonttools>=4.22.0 in c:\users\rajen\
anaconda3\envs\notebook\lib\site-packages (from matplotlib) (4.49.0)
Requirement already satisfied: kiwisolver>=1.3.1 in c:\users\rajen\
```

```
anaconda3\envs\notebook\lib\site-packages (from matplotlib) (1.4.5)
Requirement already satisfied: numpy<2,>=1.21 in c:\users\rajen\
anaconda3\envs\notebook\lib\site-packages (from matplotlib) (1.26.4)
Requirement already satisfied: packaging>=20.0 in c:\users\rajen\
anaconda3\envs\notebook\lib\site-packages (from matplotlib) (23.1)
Requirement already satisfied: pillow>=8 in c:\users\rajen\anaconda3\
envs\notebook\lib\site-packages (from matplotlib) (10.2.0)
Requirement already satisfied: pyparsing>=2.3.1 in c:\users\rajen\
anaconda3\envs\notebook\lib\site-packages (from matplotlib) (3.1.1)
Requirement already satisfied: python-dateutil>=2.7 in c:\users\rajen\
anaconda3\envs\notebook\lib\site-packages (from matplotlib) (2.8.2)
Requirement already satisfied: six>=1.5 in c:\users\rajen\anaconda3\
envs\notebook\lib\site-packages (from python-dateutil>=2.7-
>matplotlib) (1.16.0)
Note: you may need to restart the kernel to use updated packages.
```

Distplots Distplot stands for distribution plot, it takes as input an array and plots a curve corresponding to the distribution of points in the array.

- Plotting a Distplot:

```
import matplotlib.pyplot as plt
import seaborn as sns

sns.distplot([0, 1, 2, 3, 4, 5])

plt.show()
```

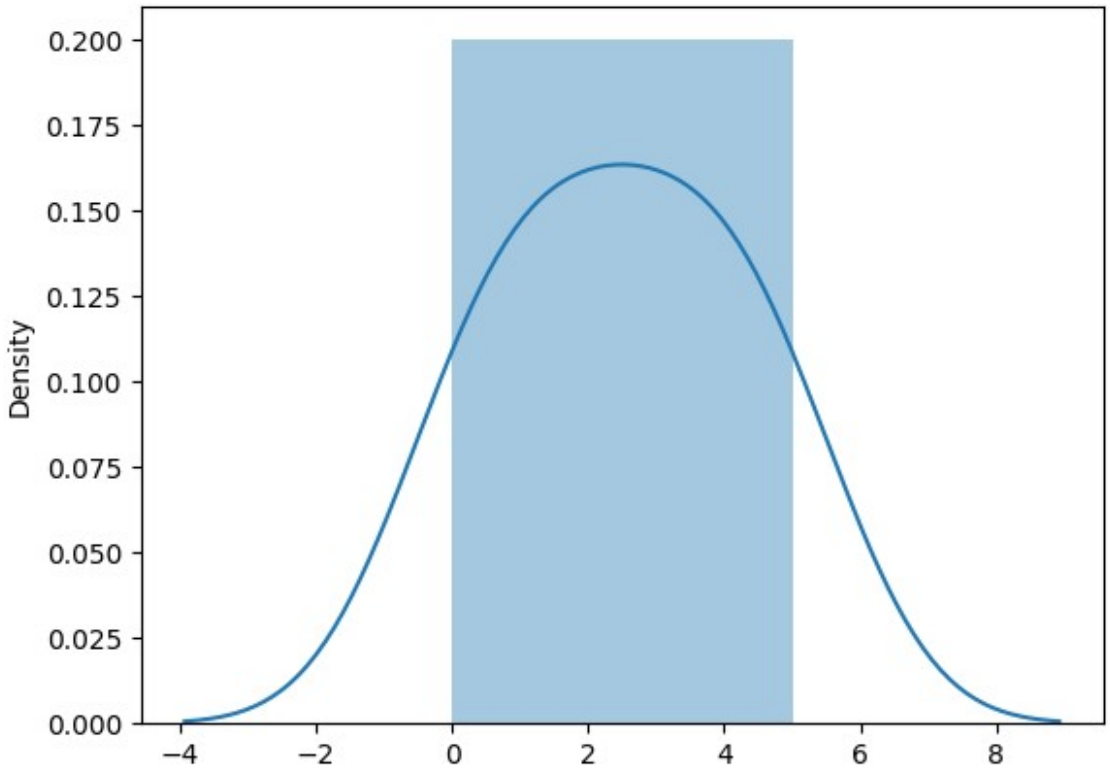
```
C:\Users\rajen\AppData\Local\Temp\ipykernel_25820\174120514.py:4:
UserWarning:

`distplot` is a deprecated function and will be removed in seaborn
v0.14.0.

Please adapt your code to use either `displot` (a figure-level
function with
similar flexibility) or `histplot` (an axes-level function for
histograms).

For a guide to updating your code to use the new functions, please see
https://gist.github.com/mwaskom/de44147ed2974457ad6372750bbe5751

  sns.distplot([0, 1, 2, 3, 4, 5])
```



- Plotting a Distplot:

```
import matplotlib.pyplot as plt
import seaborn as sns

sns.distplot([0, 1, 2, 3, 4, 5])

plt.show()
```

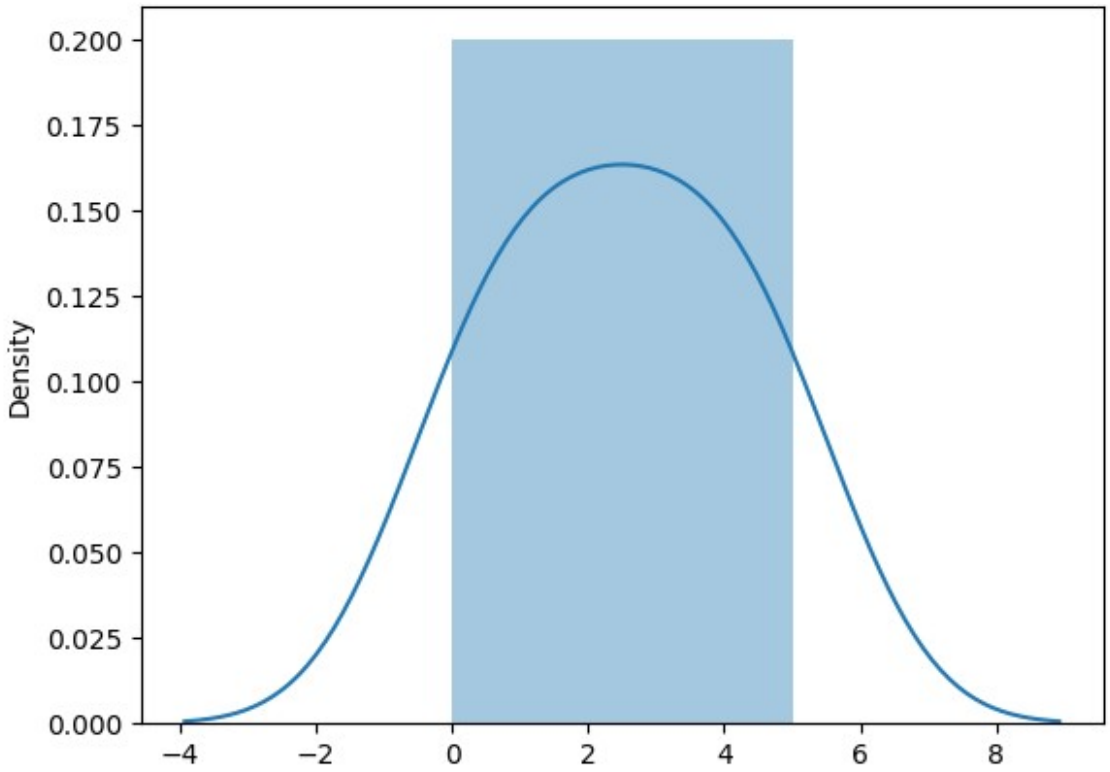
```
C:\Users\rajen\AppData\Local\Temp\ipykernel_25820\174120514.py:4:
UserWarning:

`distplot` is a deprecated function and will be removed in seaborn
v0.14.0.

Please adapt your code to use either `displot` (a figure-level
function with
similar flexibility) or `histplot` (an axes-level function for
histograms).

For a guide to updating your code to use the new functions, please see
https://gist.github.com/mwaskom/de44147ed2974457ad6372750bbe5751

  sns.distplot([0, 1, 2, 3, 4, 5])
```



Reference :

- python
- numpy
- pypl
- w3school
- scipy
- random
- random-w3schoool