Python for Data Science, Machine Learning ,Deep Learning and AI

Python for Devops--->Regular Core Python Knowledge

Concise code

Rich Libraries --->70 to 90% our libraries 10% we have to write the code

Data Science

Devops vs DataScience

Programming background--->DataScience

Admin related background, non-programming background--->devops

Numpy

Pandas

Matplotlib

seaborn

scikit learn

scipy

etc

I want to learn...parallely you people also will learn

30 days 1 month

Numpy--->Entry point from python for datascience

Numpy--->Numerical Python Library

What is the need of Numpy?

By using python we can perform mathematical operations

a = 10

b= 20

a+b
a-b
a*b
a/b

math.sqrt(10)

Data Science, Machine Learning ,Deep Learning and AI required complex mathematical operations...

1. numpy defines several functions to perform complex mathematical operations.
2. to fulfill performance gaps
  most of the numpy is implemented in C language
  superfast
3. nd array
    ---->n dimensional array or numpy array
   numpy acts as backbone for remaining libraries also.

matrix with all zeros   10X10 shape
nested list
[[0,0,0,0,0,0,0,0,,,],[],[]]

list of 100
identity matrix???

3.   Data Analysis
2 crore samples are analyzed..

100 points--->

new patient-->

History of Numpy:

Origin of Numpy --->Numeric Library
Numeric Library--->Jim Hugunin
Numpy--->Travis Oliphant and other contributors 2005
Open Source Library and Freeware

sir jim hugunin developed numpy or travis oliphant????

Q. In Which languages Numpy was written?
C and Python

Q. What is nd array in numpy:

The fundatmental data type to store our data: nd array

arrays are objects of ndarray class present in numpy module.

Array: an indexed collection of homogeneous elements

1 dimensional arrays--->Vector
2 dimensional arrays--->Matrix
..
n dimensional arrays

Application areas of Numpy?

Numpy Basic Introduction
Array Creation
  10+ ways
Attributes
How to access elements of array
  Basic Indexing
  Slice Operation
  Advanced Indexing
  Condition Based Selection
How to iterate elements of array:
  python's normal loops
  nditer()
  ndenumerate()

Arithmetic operators
Broadcasting
Array Manipulation functions
  reshape()
  resize()
  flatten()
  ravel()
  transpose()
  etc...
Matrix class
etc

Running Notes
Material
Videos for 6 months access
Rs 999

Bhavani:
durgasoftonlinetraining@gmail.com
durgasoftonline@gmail.com
99 2737 2737, 80 96969696

If you are facing any audio or video problem even in future please logout and login again

https://www.youtube.com/watch?v=-ffFPJlq7JA

How to install numpy:
----
2 ways

1st way: Anaconda--->flavour of python

2nd way: on top of python
pip install numpy

Performance Test:

Numpy vs Normal Python :

```
import numpy as np
from datetime import datetime

a = np.array([10,20,30])
b = np.array([1,2,3])

#dot product: A.B=10X1 + 20X2 +30X3=140

#traditional python code
```

```
def dot_product(a,b):
        result = 0
        for i,j in zip(a,b):
                result = result + i*j
        return result

before = datetime.now()
for i in range(1000000):
        dot_product(a,b)
after = datetime.now()

print('The Time taken by traditonal python:',after-before)

#numpy library code
before = datetime.now()
for i in range(1000000):
        np.dot(a,b) # this is from numpy
after = datetime.now()

print('The Time taken by Numpy Library:',after-before)
```

Array:

An indexed collection of homogeneous data elements.

C/C++/Java

How to create arrays in python:

----

inbuilt arrays concept is not there in python.

2 ways

1. By using array module
2. By using numpy module

1. By using array module:

```
-------------------------
import array
a = array.array('i',[10,20,30]) # i represents type: int array
print(type(a))
print(a)
print('Elements one by one:')
for x in a:
	print(x)
```

```
D:\durgaclasses>py test.py
<class 'array.array'>
array('i', [10, 20, 30])
Elements one by one:
10
20
30
```

Note: array module is not recommended because much library support is not available.

2. numpy module:

```
-----------------
import numpy
a = numpy.array([10,20,30])
print(type(a)) #<class 'numpy.ndarray'>
print(a)
print('Elements one by one:')
for x in a:
	print(x)
```

```
D:\durgaclasses>py test.py
<class 'numpy.ndarray'>
[10 20 30]
Elements one by one:
10
20
30
```

Python's List vs numpy ndarray:
----

1. Similarities:
----
1. Both can be used to store data
2. The order will be preserved in both. Hence indexing and slicing concepts are applicable.
3. Both are mutable, ie we can change the content.

2. Differences:
----
1. list is python's inbuilt type. we have to install and import numpy explicitly.

2. List can contain heterogeneous elements. But array contains only homogeneous elements.

3. On list, we cannot perform vector operations. But on ndarray we can perform vector operations.

4. Arrays consume less memory than list.

```
import numpy as np
import sys
l =
[10,20,30,40,50,60,70,80,90,100,10,20,30,40,50,60,70,80,90,100,10,20,30,40,50,
60,70,80,90,100]
a =
np.array([10,20,30,40,50,60,70,80,90,100,10,20,30,40,50,60,70,80,90,100,10,20,
30,40,50,60,70,80,90,100])
print('The Size of list:',sys.getsizeof(l))
print('The Size of ndarray:',sys.getsizeof(a))
```

```
The Size of list: 296
The Size of ndarray: 224
```

5. Arrays are superfast when compared with list.
6. Numpy arrays are more convenient to use while performing complex mathematical operations

https://drive.google.com/drive/folders/1asCu9DPBttM3wI44uFrWh3qXwlgzZBPn?usp=sharing

https://www.youtube.com/watch?v=-ffFPJlq7JA
https://www.youtube.com/watch?v=sFMY8TGBFto

Basic Introduction to Numpy
----

ndarray

ndarray vs python list
numpy vs python

How to create Numpy Arrays:
----
1. array()
2. arange()
3. linspace()
4. zeros()
5. ones()
6. full()
7. eye()
8. identity()
9. empty()
10. numpy.random library

    1. randint()
    2. rand()
    3. uniform()
    4. randn()
    5. normal()
    6. shuffle()

etc

1. Creation of numpy arrays by using array();

For the given list or tuple

1-D array:

```
>>> l = [10,20,30]
>>> type(l)
<class 'list'>
>>> a = np.array(l)
>>> type(a)
<class 'numpy.ndarray'>
>>> a
array([10, 20, 30])
```

Note:
a.ndim--->To know dimension of ndarray
a.dtype--->To know data type of elements

2-D array creation:

[[10,20,30],[40,50,60],[70,80,90]] --->Nested List

```
>>> a = np.array([[10,20,30],[40,50,60],[70,80,90]])
>>> type(a)
<class 'numpy.ndarray'>
>>> a.ndim
2
>>> a
array([[10, 20, 30],
       [40, 50, 60],
       [70, 80, 90]])
>>> a.ndim
2
>>> a.shape
(3, 3)
>>> a.size
9
```

eg-3: 1-D array from the tuple:

```
>>> a = np.array(('durga','ravi','shiva'))
>>> a
array(['durga', 'ravi', 'shiva'], dtype='<U5')
>>> type(a)
<class 'numpy.ndarray'>
>>> a.ndim
1
>>> a.shape
(3,)
>>> a.size
3
```

Note: Array contains only homogeneous elements.
if list contains heterogeneous elements: upcasting will be performed.

```
>>> a = np.array([10,20,10.5])
>>> a
array([10. , 20. , 10.5])
>>> a.dtype
dtype('float64')
```

```
>>> a=np.array([10,20,'a'])
>>> a
array(['10', '20', 'a'], dtype='<U11')
```

How to create array of a particular type:

We have to use dtype argument.

```
>>> a = np.array([10,20,30.5])
>>> a
array([10. , 20. , 30.5])
>>> a = np.array([10,20,30.5],dtype=int)
>>> a
array([10, 20, 30])
>>> a = np.array([10,20,30.5],dtype=float)
>>> a
array([10. , 20. , 30.5])
>>> a = np.array([10,20,30.5],dtype=bool)
>>> a
array([ True,  True,  True])
>>> a = np.array([10,20,30.5,0],dtype=bool)
>>> a
array([ True,  True,  True, False])

>>> a = np.array([10,20,30.5,0],dtype=complex)
>>> a
array([10. +0.j, 20. +0.j, 30.5+0.j,  0. +0.j])

>>> a = np.array([10,20,30.5],dtype=str)
>>> a
array(['10', '20', '30.5'], dtype='<U4')



>>> a = np.array([10,'durga'],dtype=int)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'durga'
```

object

int,float,str,bool,complex

How to create object type array:

Here any type of elements are allowed.

```
>>> a = np.array([10,'durga',10.5,True,10+20j],dtype=object)
>>> a
array([10, 'durga', 10.5, True, (10+20j)], dtype=object)

>>> a = np.array([10,'durga',10.5,True,10+20j])
>>> a
array(['10', 'durga', '10.5', 'True', '(10+20j)'], dtype='<U64')
>>>
```

arra90--->To create ndarray from the given list or tuple.

Creation of ndarray by using arange() function:

Python:

```
  1. range(n)--->n values from 0 to n-1
     range(4)--->0,1,2,3
  2. range(m,n)--->from m to n-1
    range(2,7)--->2,3,4,5,6

  3. range(begin,end,step)
      range(1,11,1)--->1,2,3,4,5,6,7,8,9,10
      range(1,11,2)--->1,3,5,7,9
      range(1,11,3)--->1,4,7,10
```

arange([start,] stop[, step,], dtype=None, *, like=None)

eg-1:

```
>>> a = np.arange(10)
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> a.ndim
1
>>> a.shape
(10,)
>>> a.dtype
dtype('int32')
```

eg-2:

```
>>> a = np.arange(1,11)
>>> a
array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10])
```

eg-3:

```
>>> a = np.arange(1,11,2)
>>> a
array([1, 3, 5, 7, 9])
```

```
>>> a = np.arange(1,11,3,dtype=float)
>>> a
array([ 1.,  4.,  7., 10.])
```

so no 2d array by arange()?

linspace():

----

in the specified interval , linearly spaced values

linspace(start, stop, num=50, endpoint=True, retstep=False, dtype=None, axis=0)
Return evenly spaced numbers over a specified interval.

np.linspace(0,1)

arange() vs linspace()
----
arange()--->elements will be considered in the given range based on step value.
linspace() --->The specified number of values will be considered in the given range.

zeros():

(10,)--->1-D array contains 10 elements
(5,2)--->2-D array contains 5 rows and 2 columns

2-D array means a collectio of 1-D arrays

(2,3,4)--->3-D array
3-D array contains a collection of 2-D arrays

2--->2 number of 2-D arrays
3--->The number of rows in every 2-D array

4--->The number of columns in every 2-D array
size: 24

(10,)
(5,2)
(4,3,2)

```
zeros(shape, dtype=float, order='C', *, like=None)
```

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

do we have (1,3,2) array?

1   2-D array
3 rows and 2 columns

(2,2,3,4)
4-D array means a group of 3-D arrays
2   3-D arrays are there

Every 3-D array contains 2   2-D arrays
Every 2-D array contains 3 rows and 4 columns

Total number of elements: 48

```
[
        [
                [[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]]
        ]
]
```

```
>>> np.zeros((2,2,3,4),dtype=int)
        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]]]])
```

performing some operations the result we have to store somewhere

1. ones():

Exactly same as zeros except that instead of zero array filled with value 1.
fill_value is 1

```
>>> np.ones(10)
array([1., 1., 1., 1., 1., 1., 1., 1., 1., 1.])
>>> np.ones((5,2),dtype=int)
array([[1, 1],
       [1, 1],
       [1, 1],
       [1, 1],
       [1, 1]])
>>> np.ones((2,3,4),dtype=int)
array([[[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. array()
2. arange()
3. linspace()
4. zeros()
5. ones()
6. full()

```
 full(shape, fill_value, dtype=None, order='C', *, like=None)
```

```
>>> np.full(10,2)
array([2, 2, 2, 2, 2, 2, 2, 2, 2, 2])
>>> np.full(10,4)
array([4, 4, 4, 4, 4, 4, 4, 4, 4, 4])
>>> np.full((5,4),7)
array([[7, 7, 7, 7],
       [7, 7, 7, 7],
       [7, 7, 7, 7],
       [7, 7, 7, 7],
       [7, 7, 7, 7]])
>>> np.full((2,3,4),9)
array([[[9, 9, 9, 9],
        [9, 9, 9, 9],
        [9, 9, 9, 9]],

       [[9, 9, 9, 9],
        [9, 9, 9, 9],
        [9, 9, 9, 9]]])
```

eye()
identity()

https://www.youtube.com/playlist?list=PLd3UqWTnYXOmq7EMCWvIcyPHyoOj8JClb

https://drive.google.com/drive/folders/1asCu9DPBttM3wI44uFrWh3qXwlgzZBPn?usp=sharing

full(shape, fill_value, dtype=None, order='C', *, like=None)

np.full(shape=(2,3,4),fill_value=7)
np.full((2,3,4),fill_value=7)
np.full((2,3,4),7)

zeros()
ones()
full()

eye():

To generate identity matrix

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

f(a,b)

f(a,/,b)--->We should pass values as positional arguments only

```
  f(10,20)
  f(10,b=20)
  f(a=10,b=20)--->invalid

  /--->before variables
```

```
f(a,*,b)
f(*,a,b)
```

***-->for The variables after *, we should provide values by keyword arguemnts only/**

eye(N, M=None, k=0, dtype=<class 'float'>, order='C', *, like=None)

Return a 2-D array with ones on the diagonal and zeros elsewhere.

N--->Number of rows
M--->Number of columns

```
>>> np.eye(2,3)
array([[1., 0., 0.],
      [0., 1., 0.]])
>>> np.eye(3,2)
array([[1., 0.],
      [0., 1.],
      [0., 0.]])
>>> np.eye(3)
array([[1., 0., 0.],
      [0., 1., 0.],
      [0., 0., 1.]])
>>> np.eye(3,dtype=int)
array([[1, 0, 0],
      [0, 1, 0],
      [0, 0, 1]])
>>> np.eye(5)
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.]])
>>> np.eye(5,k=1)
array([[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.]])
>>> np.eye(5,k=-3)
array([[0., 0., 0., 0., 0.],
      [0., 0., 0., 0., 0.],
      [0., 0., 0., 0., 0.],
      [1., 0., 0., 0., 0.],
      [0., 1., 0., 0., 0.]])
```

Q. Which of the following is true about returned array of eye() function?

A. It is always 2-D array.
B. The number of rows and number of columns need not be same.
C. Bydefault main diagonal contains 1s. But we can customize the diagonal which has to contain 1s.
D. All of these.

Ans: D

Q. Which of the following is true about returned array of eye() function?

A. It can be any dimensional array.
B. The number of rows and number of columns must be same.
C. only main diagonal contains 1s.
D. None of these.

Ans: D

eye always return 2d array so why A is not correct in 1st question

identity() function:

It is exactly same as eye() function except that
   1. It is always square matrix(The number of rows and number of columns always same)
   2. only main diagonal contains 1s

identity() is special case of eye()

identity(N,

sir can we get any value other than 1 in eye and identity???
No

Q. Which of the following is true about returned array of identity() function?

A. It is always 2-D array.
B. The number of rows and number of columns need not be same.

C. Bydefault main diagonal contains 1s. But we can customize the diagonal which has to contain 1s.
D. All of these.

Ans: A

Q. Which of the following is true about returned array of identity() function?

A. It can be any dimensional array.
B. The number of rows and number of columns must be same.
C. only main diagonal contains 1s.
D. None of these.

Ans: B,C

array()
arange()
linspace()
zeros()
ones()
full()
eye()
identity()
empty():
----
empty(shape, dtype=float, order='C', *, like=None)

Return a new array of given shape and type, without initializing entries.

np.empty((3,3))

zeros() vs empty():

If we required an array only with zeros then we should go for zeros().
If we never worry about data, just we required an empty array for future purpose, then we should go for empty().
The time required to create emtpy array is very very less when compared with zeros array. i.e performance wise empty() function is recommended than zeros() if we are not worry about data.
eg:

```
import numpy as np
from datetime import datetime
import sys

begin = datetime.now()
a = np.zeros((25000,300,400))
after = datetime.now()
print('Time taken by zeros:',after-begin)

a= None
begin = datetime.now()
a = np.empty((25000,300,400))
after = datetime.now()
print('Time taken by empty:',after-begin)
```

```
D:\durgaclasses>py test.py
Time taken by zeros: 0:00:00.430188
Time taken by empty: 0:00:00.056541
```

1. array()
2. arange()

3. linspace()
4. zeros()
5. ones()
6. full()
7. eye()
8. identity()

Numpy: ndarray
Scipy
Pandas: Series and DataFrame
Matplotlib--->Seaborn-->plotly

Array creation by using random library:

This library contains several functions to create nd arrays with random data.

1. randint()
2. rand()
3. uniform()
4. randn()
5. normal()
6. shuffle()
etc

1. randint():

To generate random int values in the given range

randint(low, high=None, size=None, dtype=int)

Return random integers from `low` (inclusive) to `high` (exclusive).
[low,high)

eg-1:
np.random.randint(10,20)
it will generate a single random int value in the range 10 to 19.

eg-2: To create 1-D ndarray of size 10 with random values from 1 to 8?

```
np.random.randint(1,9,size=10)
>>> np.random.randint(1,9,size=10)
array([6, 7, 6, 3, 5, 3, 3, 5, 4, 2])
>>> np.random.randint(1,9,size=10)
array([5, 3, 2, 3, 4, 7, 7, 1, 1, 6])
>>> np.random.randint(1,9,size=10)
array([8, 1, 4, 6, 6, 5, 4, 1, 4, 8])
```

eg-3: To create 2D array with shape(3,5)
np.random.randint(100,size=(3,5))

from 0 to 99 random values, 2-D array will be created

```
>>> np.random.randint(100,size=(3,5))
array([[39, 42, 90, 57, 92],
       [39, 27, 71, 96, 83],
       [47, 38, 98, 20, 82]])
```

eg-3: To create 3D array with shape(2,3,4)

3D array contains 2 2-D arrays
Each 2-D array contains 3 rows and 4 columns
np.random.randint(100,size=(2,3,4))

```
>>> np.random.randint(100,size=(2,3,4))
array([[[ 1, 50, 90, 47],
        [83, 97, 44, 85],
        [60, 16, 15, 35]],

       [[39, 52, 20,  0],
        [74, 95, 45, 40],
        [ 4, 55, 34, 67]]])
```

```
randint(low, high=None, size=None, dtype=int)
randint(low, high=None, size=None)
```

int8
int16
int32
int64

```
np.random.randint(1,11,size=(20,30))
```

```
>>> import sys
>>> a = np.random.randint(1,11,size=(20,30))
>>> sys.getsizeof(a)
2520
>>> a = np.random.randint(1,11,size=(20,30),dtype='int8')
>>> sys.getsizeof(a)
```

How to convert from one array type to another array type:

We have to use astype() method.

```
>>> a = np.random.randint(1,11,size=(20,30))
>>> a.dtype
dtype('int32')
>>> b = a.astype('float')
>>> b.dtype
dtype('float64')
```

Uniform distribution vs Normal distribution:

--

Normal Distribution is a probability distribution where probability of x is highest at centre and lowest in the ends whereas in Uniform Distribution probability of x is constant. ... Uniform Distribution is a probability distribution where probability of x is constant.

Diagram

2. rand():

It will generates random float values in the range [0,1) from uniform distribution samples.

np.random.rand() --->a single float value

```
>>> np.random.rand()
0.8120549440326994
>>> np.random.rand()
0.38273484920797385
>>> np.random.rand()
0.6674923918787643
```

```
>>> np.random.rand()
0.6895876701908819
>>> np.random.rand()
0.6462006096485643
>>> np.random.rand()
0.9098768943342903
>>> np.random.rand()
0.9362490328984621
>>> np.random.rand()
0.10852373644592084
```

1-D array:

```
np.random.rand(10)
```

2-D array:

```
>>> np.random.rand(3,5)
array([[0.33078624, 0.3070355 , 0.19368932, 0.22608363, 0.13782822],
       [0.78162618, 0.4927585 , 0.39567571, 0.15164908, 0.49992492],
       [0.54410989, 0.67688525, 0.06385654, 0.87085947, 0.00411324]])
```

Sir we will get float value in range from 0 to 1?? we can't pass range???

uniform():

----

rand()--->range is always [0,1)

uniform() --->customize range

```
uniform(low=0.0,high=1.0,size=None)
```

```
np.random.uniform()
```

```
>>> np.random.uniform(10,20)
12.132400676377454
```

n-D array creation:

```
>>> np.random.uniform(10,20,size=10)
array([19.24608642, 11.86686438, 14.68799523, 12.44187196, 12.82855603,
       12.24848812, 13.68546172, 15.9568341 , 13.57338252, 17.26628507])
>>> np.random.uniform(10,20,size=(3,5))
array([[17.48770945, 19.67756564, 14.6127878 , 15.28743195, 16.96217632],
       [10.35893547, 11.05416939, 16.49671437, 15.83282982, 15.8174664 ],
       [10.37327606, 11.9681021 , 18.19431662, 14.12503342, 11.61109241]])
>>> np.random.uniform(10,20,size=(2,3,2))
array([[[16.12847065, 13.67096846],
        [15.97964063, 19.00254701],
        [17.45192196, 16.72231574]],

       [[16.2757348 , 14.17440408],
        [17.14606001, 13.50384673],
        [10.01452353, 15.15903525]]])
```

randint()
rand()
uniform(low,high)

```
s = np.random.uniform(20,30,size=1000000)
import matplotlib.pyplot as plt
count, bins, ignored = plt.hist(s, 15, density=True)
plt.plot(bins, np.ones_like(bins), linewidth=2, color='r')
```

plt.show()

4. randn():

----

values from normal distribution with mean 0 and varience is 1

```
>>> np.random.randn(10)
array([-2.01508925, -0.28026307, -0.1646846 , -0.48833416, -0.93808559,
        1.14070496,  1.29201805, -1.35400766,  0.81779975, -0.13334964])
>>> np.random.randn(2,3)
array([[-1.26242305, -1.41742932, -0.76201615],
       [ 0.29272704,  1.14245971,  0.79194175]])
>>> np.random.randn(2,3,4)
array([[[-0.13889006,  0.35716343, -1.39591255,  0.39167841],
        [ 0.88693158,  1.03613745,  1.06677121,  0.57198922],
        [-0.28978554, -1.08459609,  1.67696806, -0.70562164]],

       [[ 0.38676079,  0.28203618,  0.12165762,  1.36922611],
        [-0.92210469, -0.83345676,  0.6830381 , -1.08446142],
        [ 0.8277217 , -0.2571676 ,  0.21365136, -0.11547937]]])
```

randint()

rand()--->uniform distribution in the range [0,1)
uniform()--->uniform distribution in our provided range

randn()--->normal distribution values with mean 0 and variance is 1
normal()--->normal distribution values with our mean and variance.

normal() function:

We can customize mean and varience .

```
normal(loc=0.0, scale=1.0, size=None)

   Draw random samples from a normal (Gaussian) distribution.
```

```
>>> np.random.normal(10,4,size=10)
array([10.84790681,  9.61363893,  8.84843827,  9.49880292,  5.75684037,
       10.35347207, 10.55850404, 13.75850698,  5.78664002, 10.21131944])
>>> np.random.normal(10,4,size=(2,3,4))
array([[[11.91237354,  9.28093298,  9.13238368, 16.32420395],
        [ 9.92394143, 11.60553826,  8.93651744, 12.34608286],
        [ 9.73972687,  9.90505171, 13.78076301, 12.88354459]],

       [[10.96347165,  9.01584661, 10.6117083 , 15.6380222 ],
        [ 4.49485441,  7.50903271,  4.92598954, 10.56220944],
        [ 7.49738784,  9.71428346,  6.42993589, 14.21750888]]])
```

```
s = np.random.normal(10,4,1000000)
import matplotlib.pyplot as plt
count, bins, ignored = plt.hist(s, 15, density=True)
plt.plot(bins, np.ones_like(bins), linewidth=2, color='r')
```

plt.show()

randint()

rand()
uniform()

randn()
normal()

Array By using random library:

An array with random int values in the given range--->np.random.randint()

An array with random float values in the range[0,1) from uniform distribution--->np.random.rand()

An array with random float values in the specified range from uniform distribution--->np.random.uniform()

An array with random float values with mean 0 and standard deviation 1 from normal distribution--->np.random.randn()

An array with random float values with specified mean and standard deviation from normal distribution--->np.random.normal()

Uniform distribution vs Normal distribution:

Normal Distribution is a probability distribution where probability of x is highest at centre and lowest in the ends whereas in Uniform Distribution probability of x is constant. ... Uniform Distribution is a probability distribution where probability of x is constant.

Diagram

randint()
rand()
uniform()
randn()
normal()

shuffle():

----

shuffle(x)

a = np.arange(9)

for 2-D arrays:

This function only shuffles the array along the first axis of a multi-dimensional array(axis-0). The order of sub-arrays is changed but their contents remains the same.

a = np.randint(1,101,size=(6,5))

```
>>> a = np.random.randint(1,101,size=(6,5))
>>> a
array([[20, 87, 85, 18, 64],
       [77, 31, 23, 80,  9],
       [42, 86, 17, 46,  7],
       [65, 89, 99, 26, 27],
       [94, 55, 61, 78,  7],
       [82, 26, 20, 16, 95]])
>>> np.random.shuffle(a)
>>> a
array([[77, 31, 23, 80,  9],
       [65, 89, 99, 26, 27],
       [82, 26, 20, 16, 95],
       [94, 55, 61, 78,  7],
       [42, 86, 17, 46,  7],
       [20, 87, 85, 18, 64]])
```

axis-0

2-D array: (3,5)
axis-0--->No of rows
axis-1--->no of columns

eg: For 3-D array
shape: (4,3,4)
4 --->2-D arrays are there(axis-0)
3 rows are there in every 2-D(axis-1)
4 columns in every 2-D array(axis-2)

Diagram

If we apply shuffle for 3-D array, then the order of 2-D arrays will be changed but not its internal content.

```
>>> a = np.arange(48).reshape(4,3,4)
>>> a
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, 30, 31],
        [32, 33, 34, 35]],

       [[36, 37, 38, 39],
        [40, 41, 42, 43],
        [44, 45, 46, 47]]])
 >>> np.random.shuffle(a)
>>> a
array([[[12, 13, 14, 15],
        [16, 17, 18, 19],
        [20, 21, 22, 23]],

       [[36, 37, 38, 39],
        [40, 41, 42, 43],
        [44, 45, 46, 47]],

       [[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [ 8,  9, 10, 11]],

       [[24, 25, 26, 27],
        [28, 29, 30, 31],
        [32, 33, 34, 35]]])
```

Summary of random library functions:

----

1. randint()--->To generate random int values in the given range.

2. rand()--->To generate uniform distributed float values in [0,1)

3. uniform()---> To generate uniform distributed float values in the given range.

4. randn()--->normal distributed float values with mean 0 and standard deviation 1.

5. normal()--->normal distributed float values with specified mean and standard deviation.

6. shuffle()-->To shuffle order of elements in the given nd array.

Numpy Array creation:

1. array()

2. arange()

3. linspace()

4. zeros()

5. ones()

6. full()

7. eye()

8. identity()

9. empty()

10 random library functions

- **randint()**
- **rand()**
- **uniform()**
- **randn()**
- **normal()**
- **shuffle()**

Array Attributes:

----

1. a.ndim--->returns the dimension of the array

2. a.shape-->Return shape of the array(10,) (2,3,4)

3. size---->To get total number of elements

4. dtype --->To get data type of array elements

5. itemsize--->4 Bytes

[10,20,30,40]

```
>>> a.shape
(4,)
```

```
>>> a = np.array([10,20,30,40])
>>> a.ndim
1
>>> a.shape
(4,)
>>> a.dtype
dtype('int32')
>>> a.size
4
>>> a.itemsize
4
```

```
>>> a = np.array([[10,20,30],[40,50,60],[70,80,90]],dtype='float')
>>> a
array([[10., 20., 30.],
     [40., 50., 60.],
     [70., 80., 90.]])
>>> a.ndim
```

```
2
>>> a.shape
(3, 3)
>>> a.size
9
>>> a.dtype
dtype('float64')
>>> a.itemsize
8
```

Numpy Data Types:

In Python : int,float,complex,bool,str etc

But in Numpy there are some extra data types also

i--->integer(int8,int16,int32,int64)
b--->boolean
u--->unsigned integer(uint8,uint16,uint32,uint64)
f--->float(float16,float32,float64)
c--->complex(complex64,complex128)
s-->String
U--->Unicode String
M-->datetime
etc

int8:

The value will be represented by 8 bits.
MSB is reserved for sign.
The range: -128 to 127

int16:

The value will be represented by 16 bits.
MSB is reserved for sign.
The range: -32768 to 32767

int32:

The value will be represented by 32 bits.
MSB is reserved for sign.
The range: -2147483648 to 2147483647

int64:

The value will be represented by 64 bits.
MSB is reserved for sign.
The range: -9223372036854775808 to 9223372036854775807

For efficient use of memory

```
>>> a = np.array([10,20,30,40])
>>> a
array([10, 20, 30, 40])
>>> import sys
>>> sys.getsizeof(a)
120
>>> a = np.array([10,20,30,40],dtype=int8)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'int8' is not defined
>>> a = np.array([10,20,30,40],dtype='int8')
>>> sys.getsizeof(a)
108
```

How to check data type of elements of an array:

--

dtype attribute

```
arrayobj.dtype

>>> a
array([10, 20, 30, 40], dtype=int8)
>>> a.dtype
dtype('int8')
```

How to create array with required data type:

--

We have to use dtype argument.

```
a = np.array([10,20,30,40],dtype='int8')
a = np.array([10,20,30,40],dtype='int16')
a = np.array([10,20,30,40],dtype='float32')


a = np.array(['a',10,10.5],dtype=int)

>>> a = np.array(['a',10,10.5],dtype=int)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'a'
```

How to convert the type of existing array:

--

1st way:

We have to use astype() method.

```
>>> a = np.array([10,20,30,40])
>>> a
array([10, 20, 30, 40])
>>> b = a.astype('float64')
>>> b
array([10., 20., 30., 40.])
>>> a.dtype
dtype('int32')
>>> b.dtype
dtype('float64')
```

2nd way:
----
float64() function
int()
float()
str()
bool()

```
>>> a = np.array([10,20,30,40])
>>> a
array([10, 20, 30, 40])

>>> a
array([10, 20, 30, 40])
>>> a.dtype
dtype('int32')
>>> f = np.float64(a)
>>> f
array([10., 20., 30., 40.])
```

```
>>> f.dtype
dtype('float64')
```

```
>>> a = np.array([10,0,20,0,30])
>>> a
array([10,  0, 20,  0, 30])
>>> a.dtype
dtype('int32')
>>> x = np.bool(a)
<stdin>:1: DeprecationWarning: `np.bool` is a deprecated alias for the builtin
`bool`. To silence this warning, use `bool` by itself. Doing this will not modify
any behavior and is safe. If you specifically wanted the numpy scalar type, use
`np.bool_` here.
Deprecated in NumPy 1.20; for more details and guidance:
https://numpy.org/devdocs/release/1.20.0-notes.html#deprecations
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: The truth value of an array with more than one element is
ambiguous. Use a.any() or a.all()
>>> x = np.bool_(a)
>>> x
array([ True, False,  True, False,  True])
```

Numpy Introduction
Creation of Numpy Arrays
Array Attributes
Data Types
How to get/access elements of Numpy Array:
--

1. Indexing--->only one element
2. Slicing--->group of elements
3. Advanded Indexing

1. Indexing:

By using index, we can get/access single element of the array.
Zero Based indexing. ie the index of first element is 0
supports both +ve and -ve indexing.

From 1-D array:
----
a[index]

```
a = np.array([10,20,30,40,50])

>>> a = np.array([10,20,30,40,50])
>>> a
array([10, 20, 30, 40, 50])
>>> a[0]
10
>>> a[1]
20
>>> a[-1]
50

>>> a[10]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: index 10 is out of bounds for axis 0 with size 5
```

Accessing elements from 2D array:

----
a[rowindex][columnindex]

```
a = np.array([[10,20,30],[40,50,60]])
```

```
>>> a = np.array([[10,20,30],[40,50,60]])
>>> a
array([[10, 20, 30],
       [40, 50, 60]])
```

To access 50:
----
a[1][1]
a[-1][-2]
a[1][-2]
a[-1][1]

To access 30:
----
a[0][2]
a[-2][-1]
a[0][-1]
a[-2][2]

Accessing elements of 3-D array:

3-D array means collection of 2-D arrays

(2,3,4)

a[i][j][k]

i--->represents which 2-D array(index of 2-D array) | can be either +ve or -ve

j--->represents row index in that 2-D array | can be either +ve or -ve

k--->represents column index in that 2-D array | can be either +ve or -ve

a[0][1][2]

0 indexed 2-D array
In that 2-D array 1 indexed row and 2 indexed column

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

Shape: (2,3,3)
a[i][j][k]

i--->represents which 2-D array(index of 2-D array) | can be either +ve or -ve

j--->represents row index in that 2-D array | can be either +ve or -ve

k--->represents column index in that 2-D array | can be either +ve or -ve

3-D array contains 3-axeses

axis-0--->represents the number of 2-D arrays
axis-1--->Row index
axis-2--->column index

Accessing elements of 4-D array:

(2,3,4,5)

4-D array contains multiple 3-D arrays
Every 3-D array contains multiple 2-D arrays
Every 2-D array contains rows and columns

(i,j,k,l)
(2,3,4,5)

2--->represents the number of 3-D arrays
3--->Every 3-D array contains 3 2-D arrays
Every 2-D array contains 4 rows and 5 columns

a[i][j][k][l]

i--->Represents which 3-D array
j--->In that 3-D array which 2-D array
k--->row index
l-->column index

a = np.arange(1,121).reshape(2,3,4,5)

Accessing elements of ndarray By using slice operator:

--

Python's slice operator:

Slice -->small piece|part

l = [10,20,30,40,50,60,70]

syntax-1:
l[begin:end]
  It returns elements from begin index to end-1 index

l[2:6]--->from 2nd index to 5th index
l[-6:-2]--->From -6 index to -3 index

If we are not specifying begin index then default is 0
l[:3] --->from 0 index to 2nd index

If we are not specifying end index then default is upto last
l[2:]

l[:]--->complete list

Note: slice operator won't raise any IndexError

```
>>> l[2:10000]
[30, 40, 50, 60, 70, 80, 90, 100]

>>> l
[10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
>>> l[2:10000]
[30, 40, 50, 60, 70, 80, 90, 100]
>>> l[10000:20000]
[]
```

Syntax-2:
l[begin:end:step]
Default value for step is: 1

l[2:8]--->from 2nd index to 7th index
l[2:8:1]--->from 2nd index to 7th index
l[2:8:2]--->from 2nd index to 7th index, every 2nd element will be considered
l[2:8:3]--->from 2nd index to 7th index, every 3rd element will be considered

```
>>> l = [10,20,30,40,50,60,70,80,90,100]
>>> l[2:8]
[30, 40, 50, 60, 70, 80]
>>> l[2:8:1]
[30, 40, 50, 60, 70, 80]
>>> l[2:8:2]
[30, 50, 70]
>>> l[2:8:3]
[30, 60]
>>> l[::3]
[10, 40, 70, 100]
```

Note:
1. For begin,end and step we can take both positive and negative values.
But for step we cannot take zero

```
>>> l[2:7:0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: slice step cannot be zero
```

If step value is +ve then we have to consider elements from begin to end-1 in forward direction.

If step value is -ve then we have to consider elements from begin to end+1 in backward direction.

2. In forward direction, if end value is 0 then result is always empty.
3. In backward direction, if end value is -1 then result is always empty.

4. All three parameters are optional.

l[::-1]--->Total list in reverse[100,90,....0]

```
>>> l[::-1]
[100, 90, 80, 70, 60, 50, 40, 30, 20, 10]
>>> l[::-2]
[100, 80, 60, 40, 20]
```

Slice Operator on 1-D Numpy Array:
----
a[begin:end:step]

Same rules of python's slice operator are applicable here
a[2:5]

```
>>> a = np.arange(10,101,10)
>>> a
array([ 10,  20,  30,  40,  50,  60,  70,  80,  90, 100])
>>> a[2:5]
array([30, 40, 50])
>>> a[::1]
array([ 10,  20,  30,  40,  50,  60,  70,  80,  90, 100])
>>> a[::-1]
array([100,  90,  80,  70,  60,  50,  40,  30,  20,  10])
```

```
>>> a[::-2]
array([100,  80,  60,  40,  20])
```

a[2:5] --->from 2nd index to 4th index

a[5:-8] --->from begin to end-1
        --->from 5th to -9th index

Slice operator on 2-D arrays:
----
a[row,column]
a[begin:end:step, begin:end:step]

```
a = np.array([[10,20],[30,40],[50,60]])
>>> a[0:1,:]
array([[10, 20]])
```

a[::2,:]

a[0:2,1:2]
a[:2,1:]

```
>>> a = np.array([[10,20],[30,40],[50,60]])
>>> a
array([[10, 20],
      [30, 40],
      [50, 60]])
```

```
>>> a[0:1,:]
array([[10, 20]])
>>> a[0,:]
array([10, 20])
>>> a[::2,:]
array([[10, 20],
      [50, 60]])
>>> a[0:2,1:2]
array([[20],
      [40]])
>>> a[:2,1:]
array([[20],
      [40]])
```

eg-2:

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

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

```
>>> a[:,0:2]
array([[ 1,  2],
       [ 5,  6],
       [ 9, 10],
       [13, 14]])
>>> a[:,::2]
array([[ 1,  3],
       [ 5,  7],
       [ 9, 11],
       [13, 15]])
>>> a[1:3,1:3]
array([[ 6,  7],
       [10, 11]])
>>> a[::3,::3]
array([[ 1,  4],
       [13, 16]])
```

Slice Operator on 3D array:

(2,3,4)

2 --->number of 2D arrays

3--->The number of rows

4--->The number of columns

```
l = [[[1,2,3,4],[5,6,7,8],[9,10,11,12]],[[13,14,15,16],[17,18,19,20],[21,22,23,24]]]
a = np.array(l)
```

a[i,j,k]

a[begin:end:step,begin:end:step,begin:end:step]

a[:,:,0:1]
a[:,:,:1]

a[:,0:1,:]
a[:,:1,:]

a[:,::2,:]

begin:end:step

::2

a[:,0:2,1:3]

a[:,::2,::3]

To use slice operator, compulsory elements should be in order. We cannot select elements which are out of order. ie we cannot select arbitrary elements.

Advanced Indexing:

By using index, we can access only one element at a time.
a[i], a[i][j], a[i][j][k]

By using slice operator we can access multiple elements at a time, but all elements should be in order.

a[begin:end:step]
a[begin:end:step,begin:end:step]
a[begin:end:step,begin:end:step,begin:end:step]

To access arbitraty elements we should go for Advanced Indexing.

Accessing multiple arbitrary elements:

1-D array:
----
array[x]
x can be either ndarray or list, which represents required indexes.

a = np.arange(10,101,10)

[2,4,5,8]

1st way:
----
create ndarray with required indices.
indices = np.array([2,4,5,8])

pass this indices array as argument to original array
a[indices]

```
>>> a
array([ 10,  20,  30,  40,  50,  60,  70,  80,  90, 100])
```

```
>>> indices
array([2, 4, 5, 8])
>>> a[indices]
array([30, 50, 60, 90])
```

2nd way:

```
l = [2,4,5,8]
a[l]

>>> l = [2,4,5,8]
>>> a[l]
array([30, 50, 60, 90])


>>> a[[0,4,6,9]]
array([ 10,  50,  70, 100])


a[[0,9,4,6]]
```

Accessing elements of 2-D array:

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

Syntax:

```
a[[row_indices],[column_indices]]
a[[0,1,2,3],[0,1,2,3]]
```

It select elements from (0,0),(1,1),(2,2) and (3,3)

L-shape

```
>>> a[[0,1,2,3,3,3,3],[0,0,0,0,1,2,3]]
array([ 1,  5,  9, 13, 14, 15, 16])
```

Note: In advanced indexing, the required elements will be selected based on indices and with those elements a 1-D array will be created. The result of advanced indexing is always 1-D array only.

Observations:

----

1. a[[0,1,2],[0,1]]

```
IndexError: shape mismatch: indexing arrays could not be broadcast together with shapes (3,) (2,)
```

2. a[[0,1],[0,1,2]]

```
IndexError: shape mismatch: indexing arrays could not be broadcast together with shapes (2,) (3,)
```

3. a[[0,1,2],[0]]

```
>>> a[[0,1,2],[0]]
array([1, 5, 9])
```

4. a[[0],[0,1,2]]

>>> a[[0],[0,1,2]] --->(0,0),(0,1),(0,2)
array([1, 2, 3])

Accessing elements of 3-D array:
----

l = [[[1,2,3,4],[5,6,7,8],[9,10,11,12]],[[13,14,15,16],[17,18,19,20],[21,22,23,24]]]
a = np.array(l)

a[i][j][k]
  i represents the index of 2-D array
  j represents row index
  k represents column index

Syntax:
----
a[[indices of 2d array],[row indices],[column indices]]

a[[0,1],[1,1],[2,1]]
The selected elements will be present at: (0,1,2) and (1,1,1)

To access elements from 1-D array:
----
a[x]
  x can be either ndarray or list of indices

2. for 2-D array:
a[[row_indices],[column_indices]]

3. For 3-D array:

a[[indices of 2-D array],[row_indices],[column_indices]]

Condition based selection:

We can select elements based on some condition also.

Syntax: array[boolean_array]

in the boolean array, where ever True present, the corresponding value will be selcted.

```
a = np.array([10,20,30,40])
```

```
>>> a = np.array([10,20,30,40])
>>> a
array([10, 20, 30, 40])
>>> boolean_array=np.array([True,False,False,True])
>>> boolean_array
array([ True, False, False,  True])
>>> a[boolean_array]
array([10, 40])
```

a = np.array([10,20,30,40])
Select elements which are greater than 25
[False,False,True,True]

```
>>> a>25
```

```
array([False, False,  True,  True])
>>> b_a = a>25
>>> a[b_a]
array([30, 40])


>>> a[a>25]
array([30, 40])


a = np.array([10,-5,20,40,-3,-1,75])
```

1. Select all elements which are <0?

```
a[a<0]



>>> a
array([10, -5, 20, 40, -3, -1, 75])
>>> a[a<0]
array([-5, -3, -1])
>>> a[a>0]
array([10, 20, 40, 75])
>>> a[a%2==0]
array([10, 20, 40])
>>> a[a%2!=0]
array([-5, -3, -1, 75])
>>> a[a%5==0]
array([10, -5, 20, 40, 75])
```

Condition based selection is applicable for 2-D array also

```
>>> a = np.arange(1,26).reshape(5,5)
>>> a
array([[ 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]])
>>> a[a%2==0]
array([ 2,  4,  6,  8, 10, 12, 14, 16, 18, 20, 22, 24])
>>> a[a%10==0]
array([10, 20])
```

Slicing vs Advanced Indexing:

Case-1: Python's slicing:

In the case of list, slice operator will create a separate copy.
If we perform any changes in one copy those changes won't be reflected in other copy.

```
>>> l = [10,20,30,40]
>>> l2=l[::]
>>> l2
[10, 20, 30, 40]
>>> l
[10, 20, 30, 40]
>>> l2
[10, 20, 30, 40]
>>> l2[1]=7777
>>> l2
[10, 7777, 30, 40]
>>> l
[10, 20, 30, 40]
>>> l[0]=8888
>>> l
[8888, 20, 30, 40]
>>> l2
[10, 7777, 30, 40]
```

Case-2: Numpy Array Slicing:

A separate copy won't be created and just we are getting view of the original copy.

Table and View
View is logical entity where as Table is physical entity.

```
a = np.arange(10,101,10)
>>> a = np.arange(10,101,10)
>>> a
array([ 10,  20,  30,  40,  50,  60,  70,  80,  90, 100])
>>> b = a[0:4]
>>> b
array([10, 20, 30, 40])
>>> a[0]=7777
>>> a
array([7777,   20,   30,   40,   50,   60,   70,   80,   90,  100])
>>> b
array([7777,   20,   30,   40])
>>> b[1]=8888
>>> b
array([7777, 8888,   30,   40])
>>> a
array([7777, 8888,   30,   40,   50,   60,   70,   80,   90,  100])
```

Case-3: Advanced Indexing and Condition Based Selection:

--

It will select required elements based on provided index or condition and with those elements a new 1-D array object will be created.

The output is always a new 1-D array only.

```
>>> a = np.arange(10,101,10)
>>> a
array([ 10,  20,  30,  40,  50,  60,  70,  80,  90, 100])
>>> b = a[[0,2,5]]
>>> b
array([10, 30, 60])
>>> a
array([ 10,  20,  30,  40,  50,  60,  70,  80,  90, 100])
>>> a[0]=7777
>>> a
array([7777,   20,   30,   40,   50,   60,   70,   80,   90,  100])
>>> b
array([10, 30, 60])
>>> b[1]=9999
>>> b
array([  10, 9999,   60])
>>> a
array([7777,   20,   30,   40,   50,   60,   70,   80,   90,  100])
```

Summary of Syntaxes:

----

1. Basic Indexing:

----

a[i],a[i][j],a[i][j][k]
a[i],a[i,j],a[i,j,k]

2. Slicing:
----
a[begin:end:step] for 1-D
a[begin:end:step,begin:end:step] for 2-D
a[begin:end:step,begin:end:step,begin:end:step] for 3-D

3. Advanced Indexing:

1.a[x] for 1-D array---> x can be ndarray or list which contains indices.
eg: a[[0,1,2]]
2. a[[row indices],[column indices]] for 2-D
3. a[[indices of 2D array],[row indices],[column indices]] for 3-D

4. Condition based Selection:
----
a[condition] eg: a[a>0]

How to Iterate elements of the ndarray:

Iteration means getting all elements one by one.

3 ways
1. By using Python's loops concept
2. By using nditer() function
3. By using ndenumerate() function

1. By using Python's loops concept:

To iterate elements of 1-D array:

```
import numpy as np
a = np.arange(10,51,10)
for x in a:
        print(x)
```

To iterate elements of 2-D array:

```
import numpy as np
a = np.array([[10,20,30],[40,50,60],[70,80,90]])
for x in a: #x is 1-D array but not scalar value
        for y in x: # y is scalar value present in 1-D array
                print(y)
```

To iterate elements of 3-D array:

```
import numpy as np
a = np.array([[[10,20],[30,40]],[[50,60],[70,80]]])
for x in a: #x is 2-D array but not scalar value
        for y in x: # y is 1-D array but not scalar value
                for z in y: # z is scalar value
                        print(z)
```

Note: To iterate elements of n-D array, we require n loops.
nditer() function

By using Numpy's nditer() function:

Advantage: For any n-D array only one loop is enough.

nditer is a class present in numpy library.
nditer() --->Creating an object of nditer class.

1-D array:

```
import numpy as np
a = np.arange(10,51,10)
for x in np.nditer(a):
        print(x)
```

2-D array:

```
import numpy as np
a = np.array([[10,20,30],[40,50,60],[70,80,90]])
for x in np.nditer(a):
        print(x)
```

3-D array:

```
import numpy as np
a = np.array([[[10,20],[30,40]],[[50,60],[70,80]]])
for x in np.nditer(a):
        print(x)
```

Iterate elements of sliced array:

```
import numpy as np
a = np.array([[10,20,30],[40,50,60],[70,80,90]])
for x in np.nditer(a[:,:2]):
        print(x)
```

Using nditer() to get elements of required data type:

We have to use op_dtypes

```
import numpy as np
a = np.array([[10,20,30],[40,50,60],[70,80,90]])
for x in np.nditer(a,flags=['buffered'],op_dtypes=['float']):
	print(x)
print(a)
```

Normal Python's loops vs nditer()

----

1. n loops are required

1. only one loop is enough

2. There is no way to specify our required dtype

2. There is a way to specify our required dtype. For this we have to use op_dtypes argument.

3. By using ndenumerate() function:

If we want to find coordinates also inaddition to elements

array indices(coordinates) and values

eg-1: Iterate elements of 1-D array:

```
import numpy as np
a = np.array([10,20,30,40,50,60,70])
for pos,element in np.ndenumerate(a):
	print(f'{element} element present at index/position:{pos}')
```

```
10 element present at index/position:(0,)
20 element present at index/position:(1,)
```

```
30 element present at index/position:(2,)
40 element present at index/position:(3,)
50 element present at index/position:(4,)
60 element present at index/position:(5,)
70 element present at index/position:(6,)
```

eg-2: Iterate elements of 2-D array:

```
import numpy as np
a = np.array([[10,20,30],[40,50,60],[70,80,90]])
for pos,element in np.ndenumerate(a):
        print(f'{element} element present at index/position:{pos}')
```

```
D:\durgaclasses>py test.py
10 element present at index/position:(0, 0)
20 element present at index/position:(0, 1)
30 element present at index/position:(0, 2)
40 element present at index/position:(1, 0)
50 element present at index/position:(1, 1)
60 element present at index/position:(1, 2)
70 element present at index/position:(2, 0)
80 element present at index/position:(2, 1)
90 element present at index/position:(2, 2)
```

python's normal loop
nditer()
ndenumerate()

Basic Indexing
Slicing
Advanced Indexing
Condition Based Selection

Arithmetic Operators:

+,-,*,/,//,%,**

10+20 --->30

20-10--->10

10*20--->200

10%2-->0

102 --->100**

/---->Normal Division Operator --->Result is always float

// --->Floor division operator

10/2 --->5.0

10/3-->3.333333

Normal division opeartor(/) always returns float value only.

Floor division operator(//) can return both int and float

If both arguments are int type then result is int

If atleast one argument is float type then result is float

10//2-->5

10//3-->3

10.0//2-->5.0

10.0//3--->3.0

15/4 --->3.75

15.0/4-->3.75

15//4--->3
15.0//4--->3.0

15.0//4--->3.0

1. Arrays with scalar
2. Arrays with Arrays

1. Arrays with scalar value:
----
scalar value means constant value like 3 or 4
+,-,*,**,%,/,//

All these operations will be performed at element level.

eg: for 1-D array

```
>>> a = np.array([10,20,30,40])
>>> a+2
array([12, 22, 32, 42])
>>> a-2
array([ 8, 18, 28, 38])
>>> a*2
array([20, 40, 60, 80])
>>> a**2
array([ 100,  400,  900, 1600], dtype=int32)
>>> a%2
```

```
array([0, 0, 0, 0], dtype=int32)
>>> a/2
array([ 5., 10., 15., 20.])
>>> a//2
array([ 5, 10, 15, 20], dtype=int32)
```

for 2-D array:

```
>>> a = np.array([[10,20,30],[40,50,60]])
>>> a
array([[10, 20, 30],
       [40, 50, 60]])
>>> a+2
array([[12, 22, 32],
       [42, 52, 62]])
>>> a-2
array([[ 8, 18, 28],
       [38, 48, 58]])
>>> a*2
array([[ 20,  40,  60],
       [ 80, 100, 120]])
>>> a/2
array([[ 5., 10., 15.],
       [20., 25., 30.]])
>>> a//2
array([[ 5, 10, 15],
       [20, 25, 30]], dtype=int32)
>>> a%2
array([[0, 0, 0],
       [0, 0, 0]], dtype=int32)
>>> a**2
array([[ 100,  400,  900],
       [1600, 2500, 3600]], dtype=int32)
```

In Python,

Anything by zero including zero/zero also result is always:
ZeroDivisionError

```
>>> 10/0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>> 0/0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
```

In Numpy we won't get ZeroDivisionError
10/0 --->Infinity(inf)
0/0---->undefined(nan--->not a number)

```
>>> a = np.arange(6)
>>> a
array([0, 1, 2, 3, 4, 5])
>>> a+2
array([2, 3, 4, 5, 6, 7])
>>> a-2
array([-2, -1,  0,  1,  2,  3])
>>> a/0
<stdin>:1: RuntimeWarning: divide by zero encountered in true_divide
<stdin>:1: RuntimeWarning: invalid value encountered in true_divide
array([nan, inf, inf, inf, inf, inf])
```

```
>>>
```

```
l = [10,20,30,40,50,60]
l/2
```

Array with Array:

```
a = np.array([1,2,3,4])
b = np.array([10,20,30,40])

>>> a
array([1, 2, 3, 4])
>>> b
array([10, 20, 30, 40])
>>> a+b
array([11, 22, 33, 44])
>>> a-b
array([ -9, -18, -27, -36])
>>> a*b
array([ 10,  40,  90, 160])
>>> b/a
array([10., 10., 10., 10.])
>>> b//a
array([10, 10, 10, 10], dtype=int32)
```

eg-2:

```
>>> a = np.array([[1,2],[3,4]])
>>> b = np.array([[5,6],[7,8]])
>>> a
array([[1, 2],
       [3, 4]])
>>> b
array([[5, 6],
       [7, 8]])
>>> a+b
array([[ 6,  8],
       [10, 12]])
>>> a-b
array([[-4, -4],
       [-4, -4]])
>>> a*b
array([[ 5, 12],
       [21, 32]])
>>> b/a
array([[5.        , 3.        ],
       [2.33333333, 2.        ]])
```

eg-3:

```
a = np.array([10,20,30,40])
b = np.array([10,20,30,40,50])

>>> a+b
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: operands could not be broadcast together with shapes (4,) (5,)
```

```
a+b ===>np.add(a,b)
a-b ===>np.subtract(a,b)
a*b ===>np.multiply(a,b)
a/b ===>np.divide(a,b)
a//b ===>np.floor_divide(a,b)
a%b ===>np.mod(a,b)
```

ab ===>np.power(a,b)**

```
>>> a
array([10, 20, 30, 40])
>>> b = np.array([10,20,30,40])
>>> a
array([10, 20, 30, 40])
>>> b
array([10, 20, 30, 40])
>>> np.add(a,b)
array([20, 40, 60, 80])
>>> np.subtract(a,b)
array([0, 0, 0, 0])
>>> np.multiply(a,b)
array([ 100,  400,  900, 1600])
>>> np.divide(a,b)
array([1., 1., 1., 1.])
>>> np.floor_divide(a,b)
array([1, 1, 1, 1], dtype=int32)
>>> np.mod(a,b)
array([0, 0, 0, 0], dtype=int32)
```

Note: The functions which operates element by element on whole array, are called universal functions(ufunc). Hence all the above functions are ufuncs.

what is the difference between np.dot() and np.multiply()?
np.dot() -->Matrix Multiplication /dot product

np.mulitply() -->Element Multiplication

Broadcasting:

Eventhough dimensions are different,shapes are different and sizes are different still some arithmetic operations are allowed. This is because of broadcasting.

Broadcasting will be performed automatically by numpy itself.

Rules for Broadcasting:

Rule-1: Make sure both arrays should have same dimension.
Add 1's in the shape of fewer dimensional array on the left side, until both arrays have same dimension.

eg-1:
Before:
(4,3)--->2D
(3,)---->1D

After:
(4,3)--->2D
(1,3)---->2D

eg-2:
Before:
(3,2,2)--->3D
(3,)---->1D

After:

(3,2,2)--->3D

(1,1,3)---->3D

Now both arrays have same dimension.

Rule-2:

If the size of 2 arrays does not match in any dimension,the array with size equal to 1 in that dimension is expanded/increases to match other size of the same dimension.

In any dimension, the sizes are not matched and neither equal to 1, then we will get error.

eg-1:
Before
(4,3)--->2D
(1,3)---->2D

After
(4,3)--->2D
(4,3)---->2D

Now dimensions, shapes and sizes are equal.

eg-2:
Before:
(3,2,2)--->3D
(1,1,3)---->3D
After:

(3,2,2)--->3D
(3,2,3)---->3D

Note: The data will be reused from the same input array.
If the rows are required then reuse existing rows.
If columns are required then reuse existing columns.

The result is always higher dimension of input arrays.

inputs: 3-D,1-D
output: 3-D

eg-1:
```
a = np.array([10,20,30,40])
b = np.array([1,2,3])
a+b
```

Rule-1:1-D and 1-D

(4,)
(3,)

```
>>> a = np.array([10,20,30,40])
>>> b = np.array([1,2,3])
>>> a+b
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: operands could not be broadcast together with shapes (4,) (3,)
```

eg-2:

```
a = np.array([10,20,30])
```

b = np.array([40])

Rule-1: satisfied
Rule-2:
(3,)
(3,)

```
>>> a = np.array([10,20,30])
>>> b = np.array([40])
>>> a+b
array([50, 60, 70])
```

eg-3:
----
a = np.array([[10,20],[30,40],[50,60]])--->2-D shape:(3,2)
b = np.array([10,20])--->1-D   shape:(2,)

Rule-1:
Before:
(3,2)
(2,)

After:
(3,2)
(1,2)

Rule-2:
Before:
(3,2)
(1,2)
After:

```
(3,2)
(3,2)
```

```
>>> a = np.array([10,20,30])
>>> b = np.array([40])
>>> a+b
array([50, 60, 70])
>>> a = np.array([[10,20],[30,40],[50,60]])
>>> b = np.array([10,20])
>>> a
array([[10, 20],
       [30, 40],
       [50, 60]])
>>> b
array([10, 20])
>>> a
array([[10, 20],
       [30, 40],
       [50, 60]])
>>> b
array([10, 20])
>>> a+b
array([[20, 40],
       [40, 60],
       [60, 80]])
>>> a-b
array([[ 0,  0],
       [20, 20],
       [40, 40]])
>>> a*b
```

```
array([[ 100,  400],
      [ 300,  800],
      [ 500, 1200]])
>>> a/b
array([[1., 1.],
      [3., 2.],
      [5., 3.]])
>>> a//b
array([[1, 1],
      [3, 2],
      [5, 3]], dtype=int32)
>>> a%b
array([[0, 0],
      [0, 0],
      [0, 0]], dtype=int32)
```

eg-4:

```
a = np.array([[10],[20],[30]])      ---->2-D, shape:(3,1)
b = np.array([10,20,30])            ---->1-D, shape:(3,)
```

Rule-1:
Before:
(3,1)
(3,)

After:
(3,1)
(1,3)

Rule-2:
Before:
(3,1)

```
(1,3)
After:
(3,3)
(3,3)

>>> a = np.array([[10],[20],[30]])
>>> b = np.array([10,20,30])
>>> a
array([[10],
       [20],
       [30]])
>>> b
array([10, 20, 30])
>>> a+b
array([[20, 30, 40],
       [30, 40, 50],
       [40, 50, 60]])
```

2. rand()
3. uniform()
4. randn()
5. normal()
6. shuffle()

Chapter-3: Array Attributes & Numpy Data Types

Chapter-4: Indexing, Slicing and Advanced Indexing

1. **Basic Indexing -->To select only one element**
2. **Slicing--->To select multiple ordered elements**
3. **Advanced Indexing--->To select multiple arbitrary elements**
4. **Condition Based Selection --->a[condition]**

Chapter-5: How to iterate elements of the nd array

1. **By using python's loops**
2. **By using numpy nditer() function**
3. **By using numpy ndenumerate() function**

Chapter-6: Arithmetic Operators

1. **Arithmetic operators for Numpy arrays with scalar**
2. **Arithmetic operators for Arrays with Arrays**

Chapter-7: Broadcasting

Array Manipulation Functions:

Chapter-8: Array Manipulation Functions

1. **reshape()**
2. **resize()**
3. **flatten()**
4. **flat variable**
5. **ravel()**
6. **transpose()**
7. **swapaxes()**

Chapter-9: Joining of multiple arrays into a single array

1. **concatenate()**
2. **stack()**

3. vstack()
4. hstack()
5. dstack()

Chapter-9: Splitting of arrays
1. split()
2. vsplit()
3. hsplit()
4. dsplit()

Chapter-10: Sorting Elements of nd arrays
Chapter-11: Searching elements of ndarray
Chapter-12: How to insert elements into ndarray?
1. insert()
2. append()
Chapter-13: How to delete elements from ndarray?

1. reshape():
----
np.reshape(array,shape)
array.reshape(shape)

reshape: shape to another shape
(10,)---->(5,2),(2,5),(1,10),(10,1)
(24,)---->(3,8)--->(2,3,4),(6,4), (2,2,2,3)

1. The Data should not be changed.
input size and output size must be matched.
otherwise ValueError
2. No change in the data.New array object won't be created.
Just we are getting view of existing object.
View but not copy.

If we perform any change in the original array, that change will be reflected to reshaped array. Viceversa.

Table(Physical)   and View(logical)

3. We can specify unknown dimension size as -1, but only once.

```
a =(12,)
b = (6,-1)===>(6,2)

b = (-1,-1)
```

```
a = 24
b = np.reshape(a,(2,3,-1))
b = np.reshape(a,(2,-1,4))
b = np.reshape(a,(-1,3,4))

b = np.reshape(a,(5,-1))
```

4. reshape(a, newshape, order='C')

C style--->Row major order(C)
Fortran Style--->Column major order(F)

eg-1:

```
>>> a = np.arange(12).reshape(3,4)
>>> a
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
>>> np.reshape(a,(12,))
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])
```

```
>>> np.reshape(a,(12,),'C')
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])
>>> np.reshape(a,(12,),'F')
array([ 0,  4,  8,  1,  5,  9,  2,  6, 10,  3,  7, 11])
```

eg-2:

```
>>> a = np.arange(24)
>>> a
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])
>>> np.reshape(a,(6,4))
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]])
>>> np.reshape(a,(6,4),'C')
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]])
>>> np.reshape(a,(6,4),'F')
array([[ 0,  6, 12, 18],
       [ 1,  7, 13, 19],
       [ 2,  8, 14, 20],
       [ 3,  9, 15, 21],
       [ 4, 10, 16, 22],
       [ 5, 11, 17, 23]])
```

Conclusions:

----
1. To reshape array without changing data.
2. The sizes must be matched.
3. We can use either numpy library function or ndarray class method.
  np.reshape()
  a.reshape()
4. It won't create a new array object, just we will get view.
5. We can use -1 in unknown dimension, but only once.
6. order: 'C','F'

resize() function:
----
output array: can be any dimension,any shape,any size

1. input size and output size need not be matched.
2. The data may be changed.
3. we will get copy but not view
4. How to get that new data:
          np.resize()--->repeat elements of input array
            a.resize()--->use zero for extra elements
4. -1 such type of story not applicable for resize()
input: (10,)
reshape: (5,-1) valid
resize: (5,-1) invalid

5. If we use ndarray class resize() method, inline modification will be happend.

reshape: view only. -1
resize:
   np.resize()--->repeat elements of input array

a.resize()--->use 0

a = np.arange(10)

sir pls explain once why np.resize() is a function and other one is a method

Difference between numpy.resize() and ndarray.resize():

Difference between reshape() and resize():
--
sir ndarray.resize it will not create new array right

1. reshape()
2. resize()

3. flatten():

1-D,2-D,3-D,n-D

1. convert any n-D array to 1-D array.
2. It is method present in ndarray class but not numpy library function.
a.flatten()--->valid
np.flatten()-->invalid
3. a.flatten(order='C')
C-style====>row major order
F-stype===>column major order

4. It will create a new array and returns it (ie copy but not view)
5. The output of flatten method is always 1D array

```
>>> a = np.arange(6).reshape(3,2)
>>> a
array([[0, 1],
      [2, 3],
      [4, 5]])
>>> a.flatten()
array([0, 1, 2, 3, 4, 5])
>>> a
array([[0, 1],
      [2, 3],
      [4, 5]])
>>> a.flatten('F')
array([0, 2, 4, 1, 3, 5])
>>> a
array([[0, 1],
      [2, 3],
      [4, 5]])
>>> b = a.flatten()
>>> b
```

```
array([0, 1, 2, 3, 4, 5])
>>> a[0][0]=7777
>>> a
array([[7777,    1],
       [   2,    3],
       [   4,    5]])
>>> b
array([0, 1, 2, 3, 4, 5])
>>> a = np.arange(1,19).reshape(3,3,2)
>>> a.ndim
3
>>> a
array([[[ 1,  2],
        [ 3,  4],
        [ 5,  6]],

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

       [[13, 14],
        [15, 16],
        [17, 18]]])
>>> a
array([[[ 1,  2],
        [ 3,  4],
        [ 5,  6]],

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

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

3-D array--->2-D array

--

flat variable:

It is a 1-D iterator over the array.

This is a 'numpy.flatiter' instance

flatten()--->to convert any dimensional array to 1-D array.

1. reshape()
2. resize()
3. flatten()
4. flat variable
5. ravel() function

It is exactly same as flatten function except that it returns view but not copy.

1. To convert any n-D array to 1-D array.
2. It is method present in ndarray class and also numpy library function.
a.ravel()--->valid
np.ravel()-->valid

3. a.ravel(order='C')
C-style====>row major order
F-stype===>column major order

4. It returns view but not copy
5. The output of ravel() method is always 1D array

```
>>> a = np.arange(24).reshape(2,3,4)
>>> a
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]]])
>>> b = a.ravel()
>>> b
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])
>>> b[0]=7777
>>> b
array([7777,    1,    2,    3,    4,    5,    6,    7,    8,    9,   10,
         11,   12,   13,   14,   15,   16,   17,   18,   19,   20,   21,
         22,   23])
>>> a
array([[[7777,    1,    2,    3],
        [   4,    5,    6,    7],
        [   8,    9,   10,   11]],

       [[  12,   13,   14,   15],
        [  16,   17,   18,   19],
       [ 20,   21,   22,   23]]])
```

```
>>> a = np.arange(18).reshape(6,3)
>>> a
array([[ 0,  1,  2],
      [ 3,  4,  5],
      [ 6,  7,  8],
      [ 9, 10, 11],
      [12, 13, 14],
      [15, 16, 17]])
>>> b = np.ravel(a)
>>> b
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16,
      17])
```

Q. What is the difference between flatten() and ravel():
---

6. transpose()
7. swapaxes()

6. transpose():
----
to find transpose() of given ndarray.

np.transpose(a, axes=None)
   Reverse or permute the axes of an array; returns the modified array.

```
>>> a = np.arange(1,5).reshape(2,2)
>>> a
array([[1, 2],
      [3, 4]])
>>> np.transpose(a)
array([[1, 3],
      [2, 4]])
```

*****Note: No change in data and hence it returns only view but not copy.**

for 3-D array:

(2,3,4):
  2 --->2-D arrays
  3---->3 rows in every 2-D array
  4---->4 rows in every 2-D array
  24--->Total size

If we transpose this array:
np.transpose(a)
(4,3,2)
  4--->2-D arrays
  3--->3 rows in every 2-D array
  2--->2 columns in every 2-D array
  Total size: 24

transpose of 1-D array:
----
transpose of 1-D array will generate same array only.

```
>>> a = np.arange(6)
>>> a
array([0, 1, 2, 3, 4, 5])
>>> np.transpose(a)
array([0, 1, 2, 3, 4, 5])
```

4-D array:
----
a --->(2,3,4,5)
np.transpose(a) ---->(5,4,3,2)

axes parameter:

If we are not using axes parameter, then dimensions will be reversed.
axes parameter descirbes in which order we have to take axes.
It is very helpful for 3-D and 4-D arrays.

for 3-D array:(2,3,4)

The size of axis-0: 2
The size of axis-1: 3
The size of axis-2: 4

np.transpose(a) --->(4,3,2)

My required order is: (2,4,3)
np.transpose(a,axes=(0,2,1))

My required shape is: (3,2,4)
np.transpose(a,axes=(1,0,2))

For 2-D array:

----
a = np.array([[10,20,30],[40,50,60]])
a.shape--->(2,3)

axis-0---->number of rows
axis-1---->number of columns

np.transpose(a,axes=(0,1))
np.transpose(a,axes=(1,0))

Note: If we repeat same axis multiple times then we will get error.
for 3-D array:(2,3,4)
np.transpose(a,axes=(0,2,2))
ValueError: repeated axis in transpose
----

1. For 1-D array, there is no effect of transpose() function.
2. If we are not using axes argument, then dimensions will be reversed.
3. If we provide axes argument, then we can specify our own order of axes.
4. Repeated axis in transpose is not allowed.
5. axes argument is more helpful from 3-D array onwards but not for 2-D array.

ndarray class transpose() method:

The behaviour is exactly same as numpy library transpose() function.

a.transpose(*axes)

Returns a view of the array with axes transposed.

eg-1:
a = np.arange(24).reshape(2,3,4)

b = a.transpose()

```
>>> b = a.transpose((2,0,1))
>>> b.shape
(4, 2, 3)
```

a.T also

Note:

1. For 1-D array, there is no effect of transpose() function.
2. If we are not using axes argument, then dimensions will be reversed.
3. If we provide axes argument, then we can specify our own order of axes.
4. Repeated axis in transpose is not allowed.
5. axes argument is more helpful from 3-D array onwards but not for 2-D arrays.
6. Various possible syntaxes:

1. **numpy.transpose(a)**
2. **numpy.transpose(a,axes=(2,0,1))**
3. **ndarrayobject.transpose()**
4. **ndarrayobject.transpose(*axes)**
5. **ndarrayobject.T**

Here 1,3,5 lines are equal wrt functionality.

swapaxes()
----

input: (2,3,4)
output: (4,3,2),(3,2,4),(2,4,3),(3,4,2) etc

By transpose() function, we can interchange any number of dimensions. But if we want to interchange only two dimensions then we should go for swapaxes() function.

```
swapaxes(a, axis1, axis2)
   Interchange two axes of an array.
```

```
a: (2,3,4)
np.swapaxes(a,0,2)-->(4,3,2)
np.swapaxes(a,1,2)-->(2,4,3)
```

```
-----------------------------
>>> a = np.arange(6).reshape(3,2)
>>> a
array([[0, 1],
      [2, 3],
      [4, 5]])
>>> np.swapaxes(a,0,1)
array([[0, 2, 4],
      [1, 3, 5]])
>>> np.swapaxes(a,1,0)
array([[0, 2, 4],
      [1, 3, 5]])
```

ndarray class also contains swapaxes

```
------------------------------------
>>> a
array([[0, 1],
      [2, 3],
      [4, 5]])
>>> b = a.swapaxes(0,1)
>>> b
array([[0, 2, 4],
      [1, 3, 5]])
>>> help(np.ndarray.swapaxes)
Help on method_descriptor:

swapaxes(...)
   a.swapaxes(axis1, axis2)

   Return a view of the array with `axis1` and `axis2` interchanged.

   Refer to `numpy.swapaxes` for full documentation.
```

Difference between transpose() and swapaxes():

--

By using transpose() we can interchange any number of dimensions.
But by using swapaxes() we can interchange only two dimensions.

1. **reshape()**
2. **resize()**
3. **flatten()**
4. **flat variable**
5. **ravel()**
6. **transpose()**
7. **swapaxes()**

Joining of multiple ndarrays into a single array:

It is something like join queries in Oracle.

1. concatenate()
2. stack()
3. vstack()
4. hstack()
5. dstack()

1. joining of multiple ndarrays into a sigle array by using concatenate() function
--
Syntax:

```
concatenate(...)
   concatenate((a1, a2, ...), axis=0, out=None, dtype=None,
casting="same_kind")

   Join a sequence of arrays along an existing axis.
```

2-D array +2-D array
axis=None
These 2-D arrays will be flatten to 1-D array and then concatenation will be happend.

Rules:
------
1. We can join any number of arrays, but all arrays should be of same dimension.
2. The sizes of all axes, except concatenation axes must be matched.

3. The result of concatenation and out must have same shape.

eg-1: Concatenation of two 1-D arrays:

```
a = np.arange(4)
b = np.arange(5)

>>> np.concatenate((a,b))
array([0, 1, 2, 3, 0, 1, 2, 3, 4])
```

eg-2: concatenation of three 1-D arrays:

```
>>> np.concatenate((a,b,c))
array([0, 1, 2, 3, 0, 1, 2, 3, 4, 0, 1, 2])
```

Storing result by using out parameter:

```
>>> np.concatenate((a,b,c),out=d)
array([0, 1, 2, 3, 0, 1, 2, 3, 4, 0, 1, 2])
>>> d = np.empty(10,dtype=int)
>>> np.concatenate((a,b,c),out=d)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<__array_function__ internals>", line 5, in concatenate
ValueError: Output array is the wrong shape
```

Using dtype parameter:

We can specify the required type by using dtype parameter.

```
>>> a
```

```
array([0, 1, 2, 3])
>>> b
array([0, 1, 2, 3, 4])
>>> np.concatenate((a,b),dtype='float')
array([0., 1., 2., 3., 0., 1., 2., 3., 4.])
>>> np.concatenate((a,b),dtype='str')
array(['0', '1', '2', '3', '0', '1', '2', '3', '4'], dtype='<U11')
```

*****Note: We cannot use dtype and out simultaneously, because out array has its own dtype.**

```
>>> a = np.arange(4)
>>> b = np.arange(5)
>>> c = np.empty(9)
>>> np.concatenate((a,b),out=c,dtype='int')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<__array_function__ internals>", line 5, in concatenate
TypeError: concatenate() only takes `out` or `dtype` as an argument, but both
were provided.
```

Q. Is it possible to join 1-D array and 2-D array?
Not possible. To use concatenate() function, compulsory all input arrays must have same dimension.

```
>>> a = np.arange(5)
>>> b = np.arange(12).reshape(3,4)
>>> np.concatenate(a,b)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<__array_function__ internals>", line 5, in concatenate
TypeError: only integer scalar arrays can be converted to a scalar index
```

Joining of 2-D arrays:

For 2-D array the existing axes are:

axis-0 --->Represents the number of rows
axis-1 --->Represents the number of columns

We can perform concatenation based on either axis-0 or axis-1.
The size of all dimensions(axes) must be matched except concatenation axis.
If we are not specifying axis the default value is 0. If axis is None, then arrays will be flatten to 1-D array and then concatenation will be happend.

eg-1:

```
a = np.array([[10,20],[30,40],[50,60]])
b = np.array([[70,80],[90,100]])

>>> a
array([[10, 20],
       [30, 40],
       [50, 60]])
>>> b
array([[ 70,  80],
       [ 90, 100]])
```

----

eg-2:

```
a = np.arange(6).reshape(3,2)
b = np.arange(9).reshape(3,3)

>>> a = np.arange(6).reshape(3,2)
>>> b = np.arange(9).reshape(3,3)
>>> a
```

```
array([[0, 1],
       [2, 3],
       [4, 5]])
>>> b
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
>>> np.concatenate((a,b),axis=0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<__array_function__ internals>", line 5, in concatenate
ValueError: all the input array dimensions for the concatenation axis must
match exactly, but along dimension 1, the array at index 0 has size 2 and the
array at index 1 has size 3
>>> a
array([[0, 1],
       [2, 3],
       [4, 5]])
>>> b
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
>>> np.concatenate((a,b),axis=1)
array([[0, 1, 0, 1, 2],
       [2, 3, 3, 4, 5],
       [4, 5, 6, 7, 8]])

a = np.arange(4).reshape(2,2)
b = np.arange(4).reshape(2,2)

>>> a = np.arange(4).reshape(2,2)
>>> b = np.arange(4).reshape(2,2)
>>> a
```

```
array([[0, 1],
      [2, 3]])
>>> b
array([[0, 1],
      [2, 3]])
>>> np.concatenate((a,b),axis=0)
array([[0, 1],
      [2, 3],
      [0, 1],
      [2, 3]])
>>> np.concatenate((a,b),axis=1)
array([[0, 1, 0, 1],
      [2, 3, 2, 3]])
>>> np.concatenate((a,b),axis=None)
array([0, 1, 2, 3, 0, 1, 2, 3])
```

Concatenation of 3-D arrays:

(x,y,z)

axis-0 --->The number of 2-D arrays

axis-1 --->The number of rows in every 2-D array

axis-2 --->The number of columns in every 2-D array

eg-1:

```
>>> a = np.arange(8).reshape(2,2,2)
>>> b = np.arange(8).reshape(2,2,2)
>>> a
array([[[0, 1],
       [2, 3]],

      [[4, 5],
       [6, 7]]])
>>> b
```

```
array([[[0, 1],
        [2, 3]],

       [[4, 5],
        [6, 7]]])
>>> np.concatenate((a,b),axis=0)
array([[[0, 1],
        [2, 3]],

       [[4, 5],
        [6, 7]],

       [[0, 1],
        [2, 3]],

       [[4, 5],
        [6, 7]]])
>>> a
array([[[0, 1],
        [2, 3]],

       [[4, 5],
        [6, 7]]])
>>> b
array([[[0, 1],
        [2, 3]],

       [[4, 5],
        [6, 7]]])
>>> np.concatenate((a,b),axis=1)
array([[[0, 1],
        [2, 3],
        [0, 1],
        [2, 3]],

       [[4, 5],
        [6, 7],
        [4, 5],
        [6, 7]]])
>>> np.concatenate((a,b),axis=2)
array([[[0, 1, 0, 1],
        [2, 3, 2, 3]],

       [[4, 5, 4, 5],
        [6, 7, 6, 7]]])
>>> np.concatenate((a,b),axis=None)
array([0, 1, 2, 3, 4, 5, 6, 7, 0, 1, 2, 3, 4, 5, 6, 7])
```

a=(2,3,2)
b=(2,3,3)

axis-0--->no
axis-1--->no
axis-2---->yes

```
>>> a = np.arange(12).reshape(2,3,2)
>>> a
array([[[ 0,  1],
        [ 2,  3],
        [ 4,  5]],

       [[ 6,  7],
        [ 8,  9],
       [10, 11]]])
>>> b = np.arange(18).reshape(2,3,3)
>>> b
array([[[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8]],

      [[ 9, 10, 11],
       [12, 13, 14],
       [15, 16, 17]]])
>>> np.concatenate((a,b),axis=0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<__array_function__ internals>", line 5, in concatenate
ValueError: all the input array dimensions for the concatenation axis must
match exactly, but along dimension 2, the array at index 0 has size 2 and the
array at index 1 has size 3
>>> np.concatenate((a,b),axis=1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<__array_function__ internals>", line 5, in concatenate
ValueError: all the input array dimensions for the concatenation axis must
match exactly, but along dimension 2, the array at index 0 has size 2 and the
array at index 1 has size 3
>>> np.concatenate((a,b),axis=2)
array([[[ 0,  1,  0,  1,  2],
       [ 2,  3,  3,  4,  5],
       [ 4,  5,  6,  7,  8]],

      [[ 6,  7,  9, 10, 11],
       [ 8,  9, 12, 13, 14],
       [10, 11, 15, 16, 17]]])
>>> np.concatenate((a,b),axis=None)
```

```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11,  0,  1,  2,  3,  4,
        5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16, 17])
```

Q. a:(2,3,3) and b:(1,3,3)
axis-0 only possible

Q. (3,2,3) and (2,1,3)
Not possible to perform concatenation on any axis.
But axis=None is possible

1-D + 1-D --->1-D
2-D + 2-D --->2-D
concatenation is always based on existing axis.
All input arrays must have same dimension.
Except concatenation axis, remaining must have same size.

joining of multiple nd arrays by using stack() function:

1-D + 1-D----->2-D
2-D + 2-D---->3-D

Rules:
1. The input arrays must have same shape.
2. The resultant stacked array has one more dimension than the input arrays.
3. Joining will be happend along new axis of newly created array.

For stacking of 1-D arrays:

----

eg-1:

```
a = np.array([10,20,30])
b = np.array([40,50,60,70])
Error
```

```
>>> a = np.array([10,20,30])
>>> b = np.array([40,50,60])
>>> a
array([10, 20, 30])
>>> b
array([40, 50, 60])
>>> np.stack((a,b))
array([[10, 20, 30],
       [40, 50, 60]])
>>> np.stack((a,b))
array([[10, 20, 30],
       [40, 50, 60]])
>>> np.stack((a,b),axis=1)
array([[10, 40],
       [20, 50],
       [30, 60]])
```

eg: Stacking of 2-D arrays:

The resultant array will be: 3-D array

3-D array shape:(x,y,z)

x-->axis-0 ---->The number of 2-D arrays

y-->axis-1 ---->The number of rows in every 2-D array

z-->axis-2 ---->The number of columns in every 2-D array

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


np.stack((a,b),axis=0)
np.stack((a,b))


np.stack((a,b),axis=1)


>>> np.stack((a,b),axis=1)
array([[[ 1,  2,  3],
        [ 7,  8,  9]],

       [[ 4,  5,  6],
        [10, 11, 12]]])

z-->axis-2 ---->The number of columns in every 2-D array
np.stack((a,b),axis=2)



a = np.arange(1,7).reshape(3,2)
b = np.arange(7,13).reshape(3,2)
c = np.arange(13,19).reshape(3,2)


>>> a
array([[1, 2],
       [3, 4],
       [5, 6]])
>>> b
```

```
array([[ 7,  8],
       [ 9, 10],
       [11, 12]])
>>> c
array([[13, 14],
       [15, 16],
       [17, 18]])
```

Based on axis-0:

In 3-D array axis-0 means the number of 2-d arrays

```
np.stack((a,b,c),axis=0)
np.stack((a,b,c))
```

```
>>> np.stack((a,b,c),axis=0)
array([[[ 1,  2],
        [ 3,  4],
        [ 5,  6]],

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

       [[13, 14],
        [15, 16],
        [17, 18]]])
```

Based on axis-1:

In 3-D array, axis-1 means the number of rows.

Stacking row wise

```
np.stack((a,b,c),axis=1)

>>> np.stack((a,b,c),axis=1)
array([[[ 1,  2],
        [ 7,  8],
        [13, 14]],

       [[ 3,  4],
        [ 9, 10],
        [15, 16]],

       [[ 5,  6],
        [11, 12],
        [17, 18]]])
```

Based on axis-2:

----

in 3-D array axis-2 means the number of columns in every 2-D array.
stacking column wise

```
np.stack((a,b,c),axis=2)

>>> np.stack((a,b,c),axis=2)
array([[[ 1,  7, 13],
        [ 2,  8, 14]],

       [[ 3,  9, 15],
        [ 4, 10, 16]],

       [[ 5, 11, 17],
        [ 6, 12, 18]]])
```

Stacking of three 1-D arrays:

```
a = np.arange(4)
b = np.arange(4,8)
c = np.arange(8,12)
```

We will get 2-D array
In 2-D array avaialble axes are: axis-0 and axis-1

```
>>> a
array([0, 1, 2, 3])
>>> b
array([4, 5, 6, 7])
>>> c
array([ 8,  9, 10, 11])
```

Based on axis-0:

----

axis-0 in 2-D array means the number of rows
np.stack((a,b,c),axis=0)

```
>>> np.stack((a,b,c),axis=0)
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

Based on axis-1:

----

axis-1 in 2-D array means the number of columns
np.stack((a,b,c),axis=1)

```
>>> np.stack((a,b,c),axis=1)
array([[ 0,  4,  8],
       [ 1,  5,  9],
       [ 2,  6, 10],
       [ 3,  7, 11]])
```

Q. What is the difference between concateante() and stack() functions?
Table: concatenate() | stack()
1. Joining will be happened based on existing axis.
1. Joining will be happened based on new axis.

2. The dimension of newly created array is same as input array dimension.
2. The dimension of newly created array is one more than input array dimension.

3. To perform concatenation, all input arrays must have same dimension. The size of all dimensions except concatenation axis must be same.
3. To perform stack operation, compulsory all input arrays must have same shape.ie dimensions,sizes also needs to be same.

Joining of arrays by using vstack() function:
---
vstack--->vertical stack--->joining is always based on axis-0
For 1-D arrays--->2-D array as output.
For the remaining dimensions it acts as concatenate() along axis-0
The result of vstack() function should be atleast 2-D
For 1-D arrays, the sizes must be same.
This is equivalent to concatenation along the first axis(axis-0) after 1-D arrays of shape `(N,)` have been reshaped to `(1,N)`.

eg-1:
```
a = np.array([10,20,30,40])
b = np.array([50,60,70,80])
np.vstack((a,b))
```

eg-2:

```
a = np.arange(1,10).reshape(3,3)
b = np.arange(10,16).reshape(2,3)
np.vstack((a,b))
```

eg-3:

```
a = np.arange(1,10).reshape(3,3)
b = np.arange(10,16).reshape(3,2)
np.vstack((a,b))
```

```
ValueError: all the input array dimensions for the concatenation axis must
match exactly, but along dimension 1, the array at index 0 has size 3 and the
array at index 1 has size 2
```

eg-4: For 3-D arrays:

----

axis-0 means The number of 2-D arrays

```
a = np.arange(1,25).reshape(2,3,4)
b = np.arange(25,49).reshape(2,3,4)

>>> np.vstack((a,b))
array([[[ 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, 30, 31, 32],
        [33, 34, 35, 36]],
       [[37, 38, 39, 40],
        [41, 42, 43, 44],
        [45, 46, 47, 48]]]])
```

Joining of arrays by using hstack() function:

Exactly same as concatenate() but joining is always based on axis-1

hstack--->horizontal stack--->column wise
1-D + 1-D --->1-D
For 1-D array:

```
a = np.array([10,20,30,40])
b = np.array([50,60,70,80,90,100])
np.hstack((a,b))

array([ 10,  20,  30,  40,  50,  60,  70,  80,  90, 100])
```

eg-2: For 2-D arrays:

```
a = np.arange(1,7).reshape(3,2)
b = np.arange(7,16).reshape(3,3)


>>> a = np.arange(1,7).reshape(3,2)
>>> b = np.arange(7,16).reshape(3,3)
>>> a
array([[1, 2],
       [3, 4],
       [5, 6]])
>>> b
array([[ 7,  8,  9],
       [10, 11, 12],
       [13, 14, 15]])
>>> np.hstack((a,b))
array([[ 1,  2,  7,  8,  9],
       [ 3,  4, 10, 11, 12],
       [ 5,  6, 13, 14, 15]])
```

eg-2:

```
a = np.arange(1,7).reshape(2,3)
b = np.arange(7,16).reshape(3,3)
```

Joining of arrays by using dstack() function:

----

dstack means --->depth stack/height stack   based on axis-2

1-D and 2-D arrays will be converted to 3-D array

The result is minimum 3-D array

Stack arrays in sequence depth wise (along third axis).

This is equivalent to concatenation along the third axis after 2-D arrays
   of shape `(M,N)` have been reshaped to `(M,N,1)` and 1-D arrays of shape
   `(N,)` have been reshaped to `(1,N,1)`.

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

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

Table: Summary of joining of nd arrays:

----

1. concatenate() ---> Join a sequence of arrays along an existing axis.

2. stack()--->Join a sequence of arrays along a new axis.

3. vstack() --->Stack arrays in sequence vertically according to first axis (axis-0).

4. hstack()--->Stack arrays in sequence horizontally according to second axis(axis-1).

5. dstack()---> Stack arrays in sequence depth wise according to third axis(axis-2).

Splitting of ndarrays:

We can perform split operation by using the following functions

1. split()
2. vsplit()
3. hsplit()
4. dsplit()
5. array_split()

We will get views but not copies.

1. split():

----

array, sections_or_indices, axis

```
split(ary, indices_or_sections, axis=0)
   Split an array into multiple sub-arrays as views into `ary`
```

sections:

1. Array will be splitted into sub arrays of equal size.
2. It returns list of sub arrays

eg-1: To split 1-D array into 3 parts

```
a= np.arange(1,10)

sub_arrays = np.split(a,3)

>>> sub_arrays = np.split(a,3)
>>> sub_arrays[0]
array([1, 2, 3])
>>> sub_arrays[1]
array([4, 5, 6])
>>> sub_arrays[2]
array([7, 8, 9])
```

```
sub_arrays = np.split(a,4)

>>> sub_arrays = np.split(a,4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<__array_function__ internals>", line 5, in split
  File "C:\Python38\lib\site-packages\numpy\lib\shape_base.py", line 872, in
split
    raise ValueError(
ValueError: array split does not result in an equal division
```

eg-2: splitting of 2-D array:

----

splitting is based on axis-0 bydefault. ie row wise split(vertical split)

We can also split based on axis-1. column wise split (horizontal split)

a = np.arange(1,25).reshape(6,4)

```
>>> a = np.arange(1,25).reshape(6,4)
>>> a
array([[ 1,  2,  3,  4],
      [ 5,  6,  7,  8],
      [ 9, 10, 11, 12],
      [13, 14, 15, 16],
      [17, 18, 19, 20],
      [21, 22, 23, 24]])
>>> np.split(a,2)
[array([[ 1,  2,  3,  4],
      [ 5,  6,  7,  8],
      [ 9, 10, 11, 12]]), array([[13, 14, 15, 16],
      [17, 18, 19, 20],
      [21, 22, 23, 24]])]
>>> np.split(a,3,axis=0)
[array([[1, 2, 3, 4],
      [5, 6, 7, 8]]), array([[ 9, 10, 11, 12],
      [13, 14, 15, 16]]), array([[17, 18, 19, 20],
      [21, 22, 23, 24]])]
>>> np.split(a,6)
[array([[1, 2, 3, 4]]), array([[5, 6, 7, 8]]), array([[ 9, 10, 11, 12]]), array([[13, 14,
15, 16]]), array([[17, 18, 19, 20]]), array([[21, 22, 23, 24]])]
```

```
>>> np.split(a,4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<__array_function__ internals>", line 5, in split
  File "C:\Python38\lib\site-packages\numpy\lib\shape_base.py", line 872, in
split
    raise ValueError(
ValueError: array split does not result in an equal division
```

splitting based on axis-1:
----
axis-1 means column wise splitting (horizontal split)

```
>>> a
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12],
       [13, 14, 15, 16],
       [17, 18, 19, 20],
       [21, 22, 23, 24]])
>>> np.split(a,2,axis=1)
[array([[ 1,  2],
       [ 5,  6],
       [ 9, 10],
       [13, 14],
       [17, 18],
       [21, 22]]), array([[ 3,  4],
       [ 7,  8],
       [11, 12],
       [15, 16],
       [19, 20],
       [23, 24]])]
>>> np.split(a,4,axis=1)
```

```
[array([[ 1],
       [ 5],
       [ 9],
       [13],
       [17],
       [21]]), array([[ 2],
       [ 6],
       [10],
       [14],
       [18],
       [22]]), array([[ 3],
       [ 7],
       [11],
       [15],
       [19],
       [23]]), array([[ 4],
       [ 8],
       [12],
       [16],
       [20],
       [24]])]
>>> np.split(a,3,axis=1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<__array_function__ internals>", line 5, in split
  File "C:\Python38\lib\site-packages\numpy\lib\shape_base.py", line 872, in
split
    raise ValueError(
ValueError: array split does not result in an equal division
```

Split based on indices:

The sizes of sub arrays need not be equal.

a = np.arange(10,101,10)

Split 2-D array based on indices:
----
a = np.arange(1,13).reshape(6,2)

a = np.arange(1,19).reshape(3,6)

```
np.split(a,[1,3,5],axis=1)
np.split(a,[2,4,4],axis=1)
np.split(a,[0,2,6],axis=1)
np.split(a,[1,5,3],axis=1)
```

Splitting by vsplit():
----
vsplit means vertical split means row wise split
split is based on axis-0

vsplit(array, sections_or_indices)

To use vsplit, input array should be atleast 2-D array

a = np.arange(10)

```
>>> a = np.arange(10)
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> np.vsplit(a,2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<__array_function__ internals>", line 5, in vsplit
  File "C:\Python38\lib\site-packages\numpy\lib\shape_base.py", line 990, in
vsplit
    raise ValueError('vsplit only works on arrays of 2 or more dimensions')
ValueError: vsplit only works on arrays of 2 or more dimensions
```

For 2-D arrays:

```
>>> a = np.arange(1,13).reshape(6,2)
>>> a
array([[ 1,  2],
       [ 3,  4],
       [ 5,  6],
       [ 7,  8],
       [ 9, 10],
       [11, 12]])
>>> np.vsplit(a,2)
[array([[1, 2],
       [3, 4],
       [5, 6]]), array([[ 7,  8],
       [ 9, 10],
       [11, 12]])]
>>> np.vsplit(a,3)
[array([[1, 2],
       [3, 4]]), array([[5, 6],
       [7, 8]]), array([[ 9, 10],
       [11, 12]])]
>>> np.vsplit(a,6)
[array([[1, 2]]), array([[3, 4]]), array([[5, 6]]), array([[7, 8]]), array([[ 9, 10]]),
array([[11, 12]])]
>>> np.vsplit(a,[1,4])
[array([[1, 2]]), array([[3, 4],
       [5, 6],
       [7, 8]]), array([[ 9, 10],
       [11, 12]])]
```

splitting by hsplit():

----

split horizontally (column wise)

```
>>> a = np.arange(10)
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> np.hsplit(a,2)
[array([0, 1, 2, 3, 4]), array([5, 6, 7, 8, 9])]
>>> np.hsplit(a,10)
[array([0]), array([1]), array([2]), array([3]), array([4]), array([5]), array([6]),
array([7]), array([8]), array([9])]
>>> np.hsplit(a,3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<__array_function__ internals>", line 5, in hsplit
  File "C:\Python38\lib\site-packages\numpy\lib\shape_base.py", line 942, in
hsplit
    return split(ary, indices_or_sections, 0)
  File "<__array_function__ internals>", line 5, in split
  File "C:\Python38\lib\site-packages\numpy\lib\shape_base.py", line 872, in
split
    raise ValueError(
ValueError: array split does not result in an equal division
```

For 2-D arrays:

Based on axis-1 only

```
a = np.arange(1,13).reshape(3,4)
>>> a = np.arange(1,13).reshape(3,4)
>>> a
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12]])
>>> np.hsplit(a,2)
[array([[ 1,  2],
       [ 5,  6],
       [ 9, 10]]), array([[ 3,  4],
       [ 7,  8],
       [11, 12]])]
>>> np.hsplit(a,4)
[array([[1],
       [5],
       [9]]), array([[ 2],
       [ 6],
       [10]]), array([[ 3],
       [ 7],
       [11]]), array([[ 4],
       [ 8],
       [12]])]
>>> np.hsplit(a,3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<__array_function__ internals>", line 5, in hsplit
  File "C:\Python38\lib\site-packages\numpy\lib\shape_base.py", line 940, in
hsplit
    return split(ary, indices_or_sections, 1)
  File "<__array_function__ internals>", line 5, in split
  File "C:\Python38\lib\site-packages\numpy\lib\shape_base.py", line 872, in
split
```

```
  raise ValueError(
ValueError: array split does not result in an equal division
```

hsplit based on indices:

----

a = np.arange(10,101,10)

```
>>> a = np.arange(24).reshape(4,6)
>>> a
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]])
>>> np.hsplit(a,[2,4])
[array([[ 0,  1],
       [ 6,  7],
       [12, 13],
       [18, 19]]), array([[ 2,  3],
       [ 8,  9],
       [14, 15],
       [20, 21]]), array([[ 4,  5],
       [10, 11],
       [16, 17],
       [22, 23]])]
>>> np.hsplit(a,[1,4])
[array([[ 0],
       [ 6],
       [12],
       [18]]), array([[ 1,  2,  3],
       [ 7,  8,  9],
       [13, 14, 15],
       [19, 20, 21]]), array([[ 4,  5],
       [10, 11],
       [16, 17],
       [22, 23]])]
```

vsplit()---->split based on axis-0(rows)
hsplit()---->split based on axis-1(columns)
dsplit()---->split based on axis-2()--->3-D array

In 3-D array:
  axis-0--->number of 2-D arrays
  axis-1--->number of rows
  axis-2-->number of columns

```
>>> a = np.arange(24).reshape(2,3,4)
>>> a
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]]])
>>> np.dsplit(a,2)
[array([[[ 0,  1],
        [ 4,  5],
        [ 8,  9]],

       [[12, 13],
        [16, 17],
        [20, 21]]]), array([[[ 2,  3],
        [ 6,  7],
        [10, 11]],
       [[14, 15],
        [18, 19],
        [22, 23]]])]


>>> a = np.arange(24).reshape(2,3,4)
>>> a
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]]])
>>> np.dsplit(a,2)
[array([[[ 0,  1],
        [ 4,  5],
        [ 8,  9]],

       [[12, 13],
        [16, 17],
        [20, 21]]]), array([[[ 2,  3],
        [ 6,  7],
        [10, 11]],

       [[14, 15],
        [18, 19],
        [22, 23]]])]
>>> a
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]]])
>>> #[1,3]
>>> np.dsplit(a,[1,3])
[array([[[ 0],
        [ 4],
        [ 8]],

       [[12],
        [16],
        [20]]]), array([[[ 1,  2],
        [ 5,  6],
        [ 9, 10]],

       [[13, 14],
        [17, 18],
        [21, 22]]]), array([[[ 3],
        [ 7],
        [11]],

       [[15],
        [19],
        [23]]])]
```

splitting by using array_split():
----
split() with sections--->should be equal parts, otherwise --->error

array_split()--->sections need not be to have equal size.
4 rows--->3 parts   2,1,1

```
array_split(ary, indices_or_sections, axis=0)
    Split an array into multiple sub-arrays.

    Please refer to the ``split`` documentation.  The only difference
    between these functions is that ``array_split`` allows
    `indices_or_sections` to be an integer that does *not* equally
    divide the axis.

    For an array of length x that should be split
    into n sections, it returns x % n sub-arrays of size x//n + 1
    and the rest of size x//n.
```

10 elements --->3 sections
x=10
n=3

x % n sub-arrays of size x//n + 1--->1 sub-array of size: 4

2 sub-arrays of size: 3
4,3,3

```
>>> a = np.arange(10,101,10)
>>> a
array([ 10,  20,  30,  40,  50,  60,  70,  80,  90, 100])
>>> np.split(a,3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<__array_function__ internals>", line 5, in split
  File "C:\Python38\lib\site-packages\numpy\lib\shape_base.py", line 872, in
split
    raise ValueError(
ValueError: array split does not result in an equal division
>>> np.array_split(a,3)
[array([10, 20, 30, 40]), array([50, 60, 70]), array([ 80,  90, 100])]
```

eg-2:
11 elements   3 sections

For an array of length x that should be split
    into n sections, it returns x % n sub-arrays of size x//n + 1
    and the rest of size x//n.

x=11
n=3

x % n sub-arrays of size x//n + 1==>2 sub-arrays of size:4
rest of size x//n.===> 1 sub-array of size: 3

4,4,3

x=12
n=3
(4,4,4)

x=13
n=3
(5,4,4,)

split(),vsplit(),hsplit(),dsplit(),array_split()

eg:for 2-D array:

```
a = np.arange(24).reshape(6,4)
np.array_split(a,4)
```

For an array of length x that should be split
into n sections, it returns x % n sub-arrays of size x//n + 1
and the rest of size x//n.

x=6
n=4

x % n sub-arrays of size x//n + 1-->2 sub-arrays of size:2
rest of size x//n.--->2 sub-arrays of size:1
2,2,1,1,

```
>>> a = np.arange(24).reshape(6,4)
>>> a
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]])
>>> np.array_split(a,4)
[array([[0, 1, 2, 3],
       [4, 5, 6, 7]]), array([[ 8,  9, 10, 11],
       [12, 13, 14, 15]]), array([[16, 17, 18, 19]]), array([[20, 21, 22, 23]])]
```

Summary of split methods:
----
split()-->Split an array into multiple sub-arrays of equal size.
vsplit()-->Split array into multiple sub-arrays vertically (row wise).
hsplit()-->Split array into multiple sub-arrays horizontally (column-wise).
dsplit()--> Split array into multiple sub-arrays along the 3rd axis (depth).
array_split()-->Split an array into multiple sub-arrays of equal or near-equal size.Does not raise an exception if an equal division cannot be made.

joining
splitting

Sorting of ndarrays:
----
np.sort(a)
quicksort --->merge sort, heap sort

For numbers-->Ascending order
For Strings-->alphabetical order

eg-1: 1-D array:

```
a = np.array([70,20,60,10,50,40,30])

>>> a = np.array([70,20,60,10,50,40,30])
>>> a
array([70, 20, 60, 10, 50, 40, 30])
>>> np.sort(a)
array([10, 20, 30, 40, 50, 60, 70])
```

To sort in descending order:

1st way:

----

np.sort(a)[::-1]

```
>>> np.sort(a)[::-1]
array([70, 60, 50, 40, 30, 20, 10])
```

2nd way:

```
>>> -np.sort(-a)
array([70, 60, 50, 40, 30, 20, 10])
```

eg-2: To sort string elements in alphabetical order

a = np.array(['cat','rat','bat','vat','dog'])

```
>>> a = np.array(['cat','rat','bat','vat','dog'])
>>> a
array(['cat', 'rat', 'bat', 'vat', 'dog'], dtype='<U3')
>>> np.sort(a)
array(['bat', 'cat', 'dog', 'rat', 'vat'], dtype='<U3')
```

To sort reverse of alphabetical order:

1st way: np.sort(a)[::-1]

2nd way: -np.sort(-a)--->invalid way

eg-3: for 2-D arrays:

----

a= np.array([[40,20,70],[30,20,60],[70,90,80]])

axis-0 --->the number of rows
axis-1--->the number of columns (axis= -1)

sort operation --->sort()

Searching elements of ndarray:
----
where() function

```
where(...)
   where(condition, [x, y])
```

where() function won't return elements.
It returns indices where condition is True.

a = np.array([3,5,7,6,9,4,6,10,15])

eg-1: Find indices where the value is 7

```
>>> a = np.array([3,5,7,6,9,4,6,10,15])
>>> a
array([ 3,  5,  7,  6,  9,  4,  6, 10, 15])
>>> np.where(a==7)
(array([2], dtype=int64),)
```

eg-2: Find indices where odd numbers present in the given 1-D array?

np.where(a%2 != 0)

```
>>> np.where(a%2 != 0)
(array([0, 1, 2, 4, 8], dtype=int64),)
```

to get elements directly

```
>>> indices = np.where(a%2 != 0)
>>> a[indices]
array([ 3,  5,  7,  9, 15])
```

condition based selection:

```
>>> a[a%2!=0]
array([ 3,  5,  7,  9, 15])
```

eg-3: Find indices where element is divisible by 3?

```
>>> np.where(a%3 == 0)
(array([0, 3, 4, 6, 8], dtype=int64),)
>>> a
array([ 3,  5,  7,  6,  9,  4,  6, 10, 15])
>>> b = np.where(a%3 == 0)
>>> a[b]
array([ 3,  6,  9,  6, 15])
```

where(condition, [x, y])
 This function can perform replace operation also.

If condition satisfied that element will be replaced from x and if the condition fails that element will be replaced from y.

eg: Replace every even number with 8888 and every odd number with 7777?

```
b = np.where(a%2==0,8888,9999)
```

```
>>> a
array([ 3,  5,  7,  6,  9,  4,  6, 10, 15])
>>> b = np.where(a%2==0,8888,9999)
>>> b
array([9999, 9999, 9999, 8888, 9999, 8888, 8888, 8888, 9999])
```

eg: Every odd number present in a, replace with 9999

```
b = np.where(a%2 != 0,9999,a)
```

```
>>> a
array([ 3,  5,  7,  6,  9,  4,  6, 10, 15])
>>> b = np.where(a%2 != 0,9999)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<__array_function__ internals>", line 5, in where
ValueError: either both or neither of x and y should be given
>>> b = np.where(a%2 != 0,9999,a)
>>> b
array([9999, 9999, 9999,    6, 9999,    4,    6,   10, 9999])
```

For 2-D arrays:

We can use where() function for any n-dimensional array.

```
>>> a = np.arange(12).reshape(4,3)
>>> a
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11]])
>>> np.where(a%5==0)
(array([0, 1, 3], dtype=int64), array([0, 2, 1], dtype=int64))
```

The first ndarray represents row indices and second ndarray represents column indices. ie the required elements present at (0,0),(1,2) and (3,1) index places.

Even we can perform replacement operation also.

```
>>> np.where(a%5==0,9999,a)
array([[9999,    1,    2],
       [   3,    4, 9999],
       [   6,    7,    8],
       [   9, 9999,   11]])
```

sort()
where()

searchsorted() function:

Internally this function will use Binary Search algorithm. Hence we can call this function only for sorted arrays.
If the array is not sorted then we will get abnormal results.

searchsorted(a, v, side='left', sorter=None)
    Find indices where elements should be inserted to maintain order.

```
a = np.arange(0,31,5)
>>> a
array([ 0,  5, 10, 15, 20, 25, 30])
np.searchsorted(a,5)
```

Note: Bydefault it will always search from left hand side to identify insertion point. If we want to search from right hand side we should use side='right'

```
>>> a = np.array([3,5,7,6,7,9,4,10,15,6])
>>> a
array([ 3,  5,  7,  6,  7,  9,  4, 10, 15,  6])
>>> a = np.sort(a)
>>> a
array([ 3,  4,  5,  6,  6,  7,  7,  9, 10, 15])
>>> np.searchsorted(a,6)
3
>>> np.searchsorted(a,6,side='right')
5
```

Summary:
1. sort()--->To sort given array
2. where() --->To perform search and replace operation
3. searchsorted() --->To identify insertion point in the given sorted array.

How to insert elements into ndarrays?
----

1. insert()
2. append()

1. insert():

----

insert(arr, obj, values, axis=None)

Insert values along the given axis before the given indices.

obj--->object that defines index or indices before which the value will be inserted.

Inserting into 1-D array:

```
a = np.arange(10)
```

eg-1: To insert 7777 before index 2

```
b = np.insert(a,2,7777)
```

```
>>> b = np.insert(a,2,7777)
>>> b
array([   0,    1, 7777,    2,    3,    4,    5,    6,    7,    8,    9])
```

eg-2: To insert 7777 before index 2 and 5?

```
b = np.insert(a,[2,5],7777)
```

```
>>> b = np.insert(a,[2,5],7777)
>>> b
array([   0,    1, 7777,    2,    3,    4, 7777,    5,    6,    7,    8,
          9])
```

eg-3: To insert 7777 before index 2 and 8888 before index 5?

```
b = np.insert(a,[2,5],[7777,8888])
```

eg-4: observations

```
b = np.insert(a,[2,5],[7777,8888,9999])

ValueError: shape mismatch: value array of shape (3,)  could not be broadcast to
indexing result of shape (2,)


b = np.insert(a,[2,5,7],[7777,8888])
ValueError: shape mismatch: value array of shape (2,)  could not be broadcast to
indexing result of shape (3,)


b = np.insert(a,[2,5,5],[777,888,999])

>>> b = np.insert(a,[2,5,5],[777,888,999])
>>> b
array([  0,   1, 777,   2,   3,   4, 888, 999,   5,   6,   7,   8,   9])

b = np.insert(a,25,7777)
IndexError: index 25 is out of bounds for axis 0 with size 10
```

****Note:
Array should contain only homogeneous elements. If we are trying to insert any other type element,that element will be converted to array type automatically before insertion. If the conversion not possible then we will get error.

```
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> np.insert(a,2,123.456)
array([  0,   1, 123,   2,   3,   4,   5,   6,   7,   8,   9])
>>> np.insert(a,2,True)
array([0, 1, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
>>> np.insert(a,2,'durga')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<__array_function__ internals>", line 5, in insert
  File "C:\Python38\lib\site-packages\numpy\lib\function_base.py", line 4640,
in insert
    values = array(values, copy=False, ndmin=arr.ndim, dtype=arr.dtype)
ValueError: invalid literal for int() with base 10: 'durga'
>>> np.insert(a,2,10+20j)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<__array_function__ internals>", line 5, in insert
  File "C:\Python38\lib\site-packages\numpy\lib\function_base.py", line 4640,
in insert
    values = array(values, copy=False, ndmin=arr.ndim, dtype=arr.dtype)
TypeError: can't convert complex to int
```

Summary:

While inserting elements into 1-D array we have to take care of the following:

1. The number of indices and the number of elements should be matched.
2. Out of range index is not allowed.
3. Elements will be converted automatically to the array type

Inserting elements into 2-D arrays:
----
we should provide axis.
If we are not providing axis, then default value None,will be considered.Then the array will be flatten to 1-D array and then insertion will be happend.

eg:
```
a = np.array([[10,20],[30,40]])
```

```
np.insert(a,1,100)

>>> a = np.array([[10,20],[30,40]])
>>> np.insert(a,1,100)
array([ 10, 100,  20,  30,  40])
```

eg-2:

```
np.insert(a,1,100,axis=0)

>>> np.insert(a,1,100,axis=0)
array([[ 10,  20],
      [100, 100],
      [ 30,  40]])
```

eg-3:

```
np.insert(a,1,[100,200],axis=0)

>>> np.insert(a,1,[100,200],axis=0)
array([[ 10,  20],
      [100, 200],
      [ 30,  40]])


>>> np.insert(a,1,[100,200,300],axis=0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<__array_function__ internals>", line 5, in insert
  File "C:\Python38\lib\site-packages\numpy\lib\function_base.py", line 4652,
in insert
    new[tuple(slobj)] = values
ValueError: could not broadcast input array from shape (1,3) into shape (1,2)
```

eg:

np.insert(a,1,100,axis=1)
   To insert a new column

```
 >>> np.insert(a,1,100,axis=1)
array([[ 10, 100,  20],
       [ 30, 100,  40]])
```

```
>>> np.insert(a,1,[100,200],axis=1)
array([[ 10, 100,  20],
       [ 30, 200,  40]])
```

In 2-D array axis-0 means rows (axis -2)
In 2-D array axis-1 means columns(axis -1)

```
np.insert(a,0,[100,200],axis=-1)
>>> np.insert(a,0,[100,200],axis=-1)
array([[100,  10,  20],
       [200,  30,  40]])
```

```
>>> np.insert(a,1,[100,200,300],axis=0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<__array_function__ internals>", line 5, in insert
  File "C:\Python38\lib\site-packages\numpy\lib\function_base.py", line 4652,
in insert
    new[tuple(slobj)] = values
ValueError: could not broadcast input array from shape (1,3) into shape (1,2)
```

Appending elements to ndarray by using append():

---
insert()--->To insert elements at our required position
append()--->To insert elements at last

Syntax:
insert(array,object,values,axis)
append(array,values,axis)

append(arr, values, axis=None)
Append values to the end of an array.

eg: 1-D array

```
a = np.arange(10)
np.append(a,9999)
```

```
>>> np.append(a,9999)
array([   0,    1,    2,    3,    4,    5,    6,    7,    8,    9, 9999])
```

```
np.append(a,[10,20,30])
>>> np.append(a,[10,20,30])
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 20, 30])
```

*****Note: if we are trying to append heterogeneous element, then array elements and new element will be converted to some common type and then append will be happend.**

```
>>> a = np.arange(10)
>>> a
```

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

np.append(a,10.5)
>>> np.append(a,10.5)
array([ 0. ,  1. ,  2. ,  3. ,  4. ,  5. ,  6. ,  7. ,  8. ,  9. , 10.5])


>>> np.append(a,10.5)
array([ 0. ,  1. ,  2. ,  3. ,  4. ,  5. ,  6. ,  7. ,  8. ,  9. , 10.5])
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> np.append(a,'durga')
array(['0', '1', '2', '3', '4', '5', '6', '7', '8', '9', 'durga'],
      dtype='<U11')
>>> np.append(a,True)
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 1])
>>> np.append(a,10+20j)
array([ 0. +0.j,  1. +0.j,  2. +0.j,  3. +0.j,  4. +0.j,  5. +0.j,
        6. +0.j,  7. +0.j,  8. +0.j,  9. +0.j, 10.+20.j])
```

Appending elements to 2-D array:

----

1. We should provide axis, otherwise None will be considered and flatten to 1-D array before append operation.

*****2. If we are providing axis, then all input arrays must have same number of dimensions and same shape of provided axis.**

```
a = np.array([[10,20],[30,40]])
>>> a
array([[10, 20],
       [30, 40]])
```

```
new row: [[70,80]]

np.append(a,70)
>>> np.append(a,70)
array([10, 20, 30, 40, 70])

eg-2:
np.append(a,70,axis=0)
ValueError: all the input arrays must have same number of dimensions, but the
array
at index 0 has 2 dimension(s) and the array at index 1 has 0 dimension(s)

eg-3:
np.append(a,[70,80],axis=0)
ValueError: all the input arrays must have same number of dimensions, but the
array at index 0 has 2 dimension(s) and the array at index 1 has 1 dimension(s)


eg-4:
np.append(a,[[70,80]],axis=0)

>>> np.append(a,[[70,80]],axis=0)
array([[10, 20],
       [30, 40],
       [70, 80]])

>>> np.append(a,[[70,80],[90,100]],axis=0)
array([[ 10,  20],
       [ 30,  40],
       [ 70,  80],
       [ 90, 100]])
```

EG-5:

```
np.append(a,[[70,80]],axis=1)

ValueError: all the input array dimensions for the concatenation axis must
match exactly, but along dimension 0, the array at index 0 has size 2 and the
array at index 1 has size 1
```

```
np.append(a,[[70],[80]],axis=1)
>>> np.append(a,[[70],[80]],axis=1)
array([[10, 20, 70],
       [30, 40, 80]])
```

```
np.append(a,[[70,80],[90,100]],axis=1)
>>> np.append(a,[[70,80],[90,100]],axis=1)
array([[ 10,  20,  70,  80],
       [ 30,  40,  90, 100]])
```

Q. Consider the array?

```
>>> a = np.arange(12).reshape(4,3)
>>> a
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11]])
```

Which of the following operations will be performed successfully?

A. np.append(a,[[10,20,30]],axis=0) #valid

B. np.append(a,[[10,20,30]],axis=1) #invalid

C. np.append(a,[[10],[20],[30]],axis=0) #invalid
D. np.append(a,[[10],[20],[30],[40]],axis=1) #valid
E. np.append(a,[[10,20,30],[40,50,60]],axis=0) #valid
F. np.append(a,[[10,20],[30,40],[50,60],[70,80]],axis=1) #valid

Q. What is the difference between insert() and append() functions?
By using insert() function we can insert elements at our required index position. But by using append() function, we can add elements always at the end of ndarray.

Deletion of elements from ndarray:

We can delete elements by using delete() function of numpy module.

delete(arr, obj, axis=None)

To delete elements of 1-D array:
----
a = np.arange(10,101,10)

1. np.delete(a,3) # to delete element locating at index: 3

2. np.delete(a,[0,4,6]) ## to delete elements locating at indices:0,4,6

3. np.delete(a,np.s_[2:6]) # to delete elements locating from 2nd index to 5th index

```
>>> a = np.arange(10,101,10)
>>> a
array([ 10,  20,  30,  40,  50,  60,  70,  80,  90, 100])
>>> np.delete(a,3)
```

```
array([ 10,  20,  30,  50,  60,  70,  80,  90, 100])
>>> np.delete(a,[0,4,6])
array([ 20,  30,  40,  60,  80,  90, 100])
>>> np.delete(a,np.s_[2:6])
array([ 10,  20,  70,  80,  90, 100])
```

```
np.delete(a,range(2,6))
```

To delete elements of 2-D array:

We should provide axis. If we are not providing axis, then array will be flatten to 1-D array and then deletion will be happend.

```
a = np.arange(1,13).reshape(3,4)
```

```
>>> a = np.arange(1,13).reshape(3,4)
>>> a
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12]])
```

```
np.delete(a,1) #flatten to 1-D array and then deletion
```

```
>>> np.delete(a,1)
array([ 1,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12])
```

```
np.delete(a,0,axis=0) # to delete 0th row
np.delete(a,[0,2],axis=0) # to delete 0th row and 2nd row
```

np.delete(a,np.s_[:2],axis=0) # to delete 0th row and 1st rows

```
>>> a = np.arange(1,13).reshape(3,4)
>>> a
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12]])
>>> np.delete(a,1)
array([ 1,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12])
>>> [ 5,  6,  7,  8],
([5, 6, 7, 8],)
>>> np.delete(a,0,axis=0)
array([[ 5,  6,  7,  8],
       [ 9, 10, 11, 12]])
>>> np.delete(a,[0,2],axis=0)
array([[5, 6, 7, 8]])
>>> np.delete(a,np.s_[:2],axis=0)
array([[ 9, 10, 11, 12]])
```

```
>>> a
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12]])

np.delete(a,0,axis=1)
np.delete(a,[0,2],axis=1)
np.delete(a,np.s_[::3],axis=1)
np.delete(a,np.s_[1:],axis=1)
```

Delete elements from 3-D array:

```
a = np.arange(24).reshape(2,3,4)
>>> a
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]]])
```

axis-0--->index of 2-D array
axis-1--->Rows in every 2-D array
axis-2--->columns in every 2-D array

```
np.delete(a,3) #flatten to 1-D array and delete 3rd index element

>>> np.delete(a,3)
array([ 0,  1,  2,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16, 17,
       18, 19, 20, 21, 22, 23])

np.delete(a,0,axis=0) #To delete first 2-D array

np.delete(a,1,axis=0)

np.delete(a,1,axis=1)
```

To delete 1st indexed row in every 2-D array

```
np.delete(a,2,axis=2)
np.delete(a,[0,2],axis=2)


np.delete(a,np.s_[1:],axis=2)
```

```
>>> a = np.arange(24).reshape(2,3,4)
>>> a
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]]])
>>> np.delete(a,3)
array([ 0,  1,  2,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16, 17,
       18, 19, 20, 21, 22, 23])
>>> np.delete(a,0,axis=0)
array([[[12, 13, 14, 15],
        [16, 17, 18, 19],
        [20, 21, 22, 23]]])
>>> np.delete(a,1,axis=0)
array([[[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [ 8,  9, 10, 11]]])
>>>
>>> np.delete(a,1,axis=1)
array([[[ 0,  1,  2,  3],
        [ 8,  9, 10, 11]],

       [[12, 13, 14, 15],
        [20, 21, 22, 23]]])
>>> np.delete(a,2,axis=2)
array([[[ 0,  1,  3],
        [ 4,  5,  7],
        [ 8,  9, 11]],
       [[12, 13, 15],
        [16, 17, 19],
        [20, 21, 23]]])
>>> np.delete(a,[0,2],axis=2)
array([[[ 1,  3],
        [ 5,  7],
        [ 9, 11]],

       [[13, 15],
        [17, 19],
        [21, 23]]])
>>> np.delete(a,np.s_[1:],axis=2)
array([[[ 0],
        [ 4],
        [ 8]],

       [[12],
        [16],
        [20]]])
```

Case Study:

Q. Consider the following array:

```
>>> a = np.arange(12).reshape(4,3)
>>> a
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11]])
```

Delete last row and insert following row in that place:

[70,80,90]

Solution:

```
>>> a = np.arange(12).reshape(4,3)
>>> a
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11]])
>>> b = np.delete(a,3,axis=0)
>>> b
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
>>> np.append(b,[[70,80,90]],axis=0)
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [70, 80, 90]])
```

Summary:

----

insert()-->Insert elements into an array at specified index.

append()-->Append elements at the end of an array.

delete()--->Delete elements from an array.

Chapter-16: Matrix multiplication by using dot() function

---

ndarrays : a and b

a*b --->element level multiplication will be happend.

```
>>> a = np.array([[10,20],[30,40]])
>>> b = np.array([[1,2],[3,4]])
>>> a
array([[10, 20],
       [30, 40]])
>>> b
array([[1, 2],
       [3, 4]])
>>> a*b
array([[ 10,  40],
       [ 90, 160]])
```

If we want matrix multiplication then we should go for dot() function.

```
np.dot(a,b)
a.dot(b)
```

```
>>> import numpy as np
>>> a = np.array([[10,20],[30,40]])
>>> b = np.array([[1,2],[3,4]])
>>> a
array([[10, 20],
       [30, 40]])
>>> b
array([[1, 2],
       [3, 4]])
>>> a*b
array([[ 10,  40],
       [ 90, 160]])
>>> np.dot(a,b)
array([[ 70, 100],
       [150, 220]])
>>> a.dot(b)
array([[ 70, 100],
      [150, 220]])
>>> a
array([[10, 20],
      [30, 40]])
```

Chapter-17: Importance of matrix class in numpy library

1-D array is called--->Vector

2-D array is called-->Matrix

matrix class is specially designed class to create 2-D arrays.

How to create 2-D array:

----

1. By using ndarray
2. By using matrix class

```
class matrix(ndarray)
 |  matrix(data, dtype=None, copy=True)

data : array_like or string
 |     If `data` is a string, it is interpreted as a matrix with commas
 |     or spaces separating columns, and semicolons separating rows.
```

eg-1: Creating matrix object from string

```
a = np.matrix('col1 col2 col3;col1 col2 col3')
a = np.matrix('col1,col2,col3;col1,col2,col3')

a = np.matrix('10,20;30,40')
```

eg-2: Creating matrix object from nested list

```
a = np.matrix([[10,20],[30,40]])

>>> a = np.matrix([[10,20],[30,40]])
>>> a
matrix([[10, 20],
        [30, 40]])
```

eg-3. create a matrix from ndarray

```
>>> a = np.arange(6).reshape(3,2)
>>> type(a)
<class 'numpy.ndarray'>
>>> b = np.matrix(a)
>>> type(b)
<class 'numpy.matrix'>
>>> b
matrix([[0, 1],
        [2, 3],
        [4, 5]])
>>> a
array([[0, 1],
       [2, 3],
       [4, 5]])
```

conclusions:
----
1. matrix is child class of ndarray class. Hence all methods and attributes of ndarray class are applicable ot matrix also.

2. We can use +,*,T, for matrix objects also.**
3. In the case of ndarray, * operator performs element level multiplication.
But in case of matrix, * operator preforms matrix multiplication.

4. In the case of ndarray, ** operator performs power operation at element level.
But in the case of matrix, operator performs 'matrix' power.**

5. matrix class always meant for 2-D array only.

6. It is no longer recommended to use.

```
>>> a = np.array([[1,2],[3,4]])
>>> m = np.matrix([[1,2],[3,4]])
>>> a
array([[1, 2],
      [3, 4]])
>>> m
matrix([[1, 2],
       [3, 4]])
>>> a+a
array([[2, 4],
      [6, 8]])
>>> m+m
matrix([[2, 4],
       [6, 8]])
>>> a*a
array([[ 1,  4],
      [ 9, 16]])
>>> m*m
matrix([[ 7, 10],
        [15, 22]])
>>> a
array([[1, 2],
       [3, 4]])
>>> a**2
array([[ 1,  4],
       [ 9, 16]], dtype=int32)
>>> m**2
matrix([[ 7, 10],
        [15, 22]])
>>> a
array([[1, 2],
       [3, 4]])
>>> a.T
array([[1, 3],
       [2, 4]])
>>> m
matrix([[1, 2],
        [3, 4]])
>>> m.T
matrix([[1, 3],
        [2, 4]])
```

Differences between ndarray and matrix:

----

table

Linear Algebra functions from linalg module:

--

numpy.linalg --->To perform linear algebra operations

1. inv() ---->To find inverse of a matrix

2. matrix_power() ---->To find power of a matrix like A^n

3. det() --->To find determinant of a matrix
4. solve() --->To solve linear algebra equations
etc

1. inv() ---->To find inverse of a matrix
--

```
inv(a)
   Compute the (multiplicative) inverse of a matrix.

   Given a square matrix `a`, return the matrix `ainv` satisfying
   ``dot(a, ainv) = dot(ainv, a) = eye(a.shape[0])``.


>>> a = np.array([[1,2],[3,4]])
>>> a
array([[1, 2],
       [3, 4]])
>>> ainv = np.linalg.inv(a)
>>> ainv
array([[-2. ,  1. ],
       [ 1.5, -0.5]])
```

How to check:
----
np.dot(a,ainv) = I

```
>>> a = np.array([[1,2],[3,4]])
>>> a
array([[1, 2],
       [3, 4]])
>>> ainv = np.linalg.inv(a)
>>> ainv
array([[-2. ,  1. ],
      [ 1.5, -0.5]])
>>> i = np.eye(2)
>>> i
array([[1., 0.],
      [0., 1.]])
>>> np.dot(a,ainv)
array([[1.0000000e+00, 0.0000000e+00],
      [8.8817842e-16, 1.0000000e+00]])
>>> np.allclose(np.dot(a,ainv),i)
```

Note:
allclose(a, b, rtol=1e-05, atol=1e-08, equal_nan=False)
Returns True if two arrays are element-wise equal within a tolerance.

*****Note: We can find inverse only for square matrices, otherwise we will get error.**

```
>>> a = np.arange(10).reshape(5,2)
>>> a
array([[0, 1],
      [2, 3],
      [4, 5],
      [6, 7],
      [8, 9]])
>>> np.linalg.inv(a)
numpy.linalg.LinAlgError: Last 2 dimensions of the array must be square
```

How to find inverse of 3-D array:

3-D array contains collection of 2-D arrays
Finding inverse of 3-D array means, finding inverse for every 2-D array.

a = np.arange(8).reshape(2,2,2)
two 2-D arrays

```
>>> a = np.arange(8).reshape(2,2,2)
>>> a
array([[[0, 1],
        [2, 3]],

       [[4, 5],
        [6, 7]]])
>>> np.linalg.inv(a)
array([[[-1.5,  0.5],
        [ 1. ,  0. ]],

       [[-3.5,  2.5],
        [ 3. , -2. ]]])
```

2. matrix_power() ---->To find power of a matrix like A^n
----
np.linalg.matrix_power(a,n)

If n==0 ===>Identity Matrix
If n>0 ===>normal power operation
If n<0 ===>First inverse and then power operation for absolute value

```
matrix_power(a, n)
    Raise a square matrix to the (integer) power `n`.

    For positive integers `n`, the power is computed by repeated matrix
```

```
   squarings and matrix multiplications. If ``n == 0``, the identity matrix
   of the same shape as M is returned. If ``n < 0``, the inverse
   is computed and then raised to the ``abs(n)``.

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

>>> a = np.array([[1,2],[3,4]])
>>> a
array([[1, 2],
       [3, 4]])
>>> np.linalg.matrix_power(a,0)
array([[1, 0],
       [0, 1]])
>>> np.linalg.matrix_power(a,2)
array([[ 7, 10],
       [15, 22]])
>>> np.linalg.matrix_power(a,-2)
array([[ 5.5 , -2.5 ],
       [-3.75,  1.75]])
>>> np.dot(np.linalg.inv(a),np.linalg.inv(a))
array([[ 5.5 , -2.5 ],
       [-3.75,  1.75]])
>>> np.linalg.matrix_power(np.linalg.inv(a),2)
array([[ 5.5 , -2.5 ],
       [-3.75,  1.75]])
```

Note: We can find matrix_power only for a square matrix,otherwise we will get error.

```
>>> a = np.arange(10).reshape(5,2)
>>> a
array([[0, 1],
       [2, 3],
       [4, 5],
       [6, 7],
       [8, 9]])
>>> np.linalg.matrix_power(a,2)
numpy.linalg.LinAlgError: Last 2 dimensions of the array must be square
```

3. det() --->To find determinant of a matrix

Syntax:

det(a)

```
>>> a = np.array([[1,2],[3,4]])
>>> a
array([[1, 2],
       [3, 4]])
>>> np.linalg.det(a)
-2.0000000000000004
```

For 3X3 matrix:

```
>>> a = np.arange(9).reshape(3,3)
>>> a
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
>>> np.linalg.det(a)
0.0
```

Note: We can find determinant only for square matrices, otherwise we will get error.

```
>>> a = np.arange(10).reshape(5,2)
>>> np.linalg.det(a)
numpy.linalg.LinAlgError: Last 2 dimensions of the array must be square
```

4. solve() --->To solve linear algebra equations

```
solve(a, b)
   Solve a linear matrix equation, or system of linear scalar equations.

   Computes the "exact" solution, `x`, of the well-determined, i.e., full
   rank, linear matrix equation `ax = b`.

   Parameters
   ----------
   a : (..., M, M) array_like
      Coefficient matrix.
   b : {(..., M,), (..., M, K)}, array_like
      Ordinate or "dependent variable" values.
```

Case Study:

----

case study:

Problem:

Boys and Girls are attending Durga sir's datascience class.
For boys fee is $3 and for girls fee is $8. For a certain batch 2200 people attented and $10100 fee collected. How many boys and girls attended for that batch?

assume x is number of boys
assume y is number of girls

x+y = 2200===>x = 2200-y
3x+8y=10100

3(2200-y)+8y=10100

6600-3y+8y = 10100

5y=10100-6600
5y=3500
y=700
x=1500

x+y = 2200
3x+8y=10100

```
a = np.array([[1,1],[3,8]])
b = np.array([2200,10100])
```

```
>>> a = np.array([[1,1],[3,8]])
>>> b = np.array([2200,10100])
>>> a
array([[1, 1],
       [3, 8]])
>>> b
array([ 2200, 10100])
>>> np.linalg.solve(a,b)
array([1500.,  700.])
```

eg-2:

-4x+7y-2z = 2
x-2y+z = 3
2x-3y+z = -4

```
a = np.array([[-4,7,-2],[1,-2,1],[2,-3,1]])
b = np.array([2,3,-4])
np.linalg.solve(a,b)
array([-13.,  -6.,   4.])
```

x=-13,
y=-6
z=4

x-2y+z = 3
-13+12+4=3

1. inv() ---->To find inverse of a matrix
2. matrix_power() ---->To find power of a matrix like A^n
3. det() --->To find determinant of a matrix
4. solve() --->To solve linear algebra equations
etc

I/O Operations with Numpy:

We can save/write ndarray objects to a binary file for future purpose.
Later point of time, when ever these objects are required, we can read from that binary file.

save()--->to save/write ndarry object to a file
load() --->to read ndarray object from a file

```
save(file, arr, allow_pickle=True, fix_imports=True)
   Save an array to a binary file in NumPy ``.npy`` format.
```

load(file, mmap_mode=None, allow_pickle=False, fix_imports=True, encoding='ASCII')
Load arrays or pickled objects from ``.npy``, ``.npz`` or pickled files.

eg-1: Saving ndarray object to a file and read back:

```
import numpy as np
a = np.array([[10,20,30],[40,50,60]]) #2-D array with shape:(2,3)

#save/serialize ndarray object to a file
np.save('out.npy',a)

#load/deserialize ndarray object from a file
out_array = np.load('out.npy')
print(out_array)
```

Note:
1. The data will be stored in binary form
2. File extension should be .npy, otherwise save() function itself will add that extension.
3. By using save() function we can write only one obejct to the file. If we want to write multiple objects to a file then we should go for savez() function.

Saving mulitple ndarray objects to the binary file:
--

```
import numpy as np
a = np.array([[10,20,30],[40,50,60]]) #2-D array with shape:(2,3)
b = np.array([[70,80],[90,100]]) #2-D array with shape:(2,2)

#save/serialize ndarrays object to a file
```

```
np.savez('out.npz',a,b)

#reading ndarray objects from a file
npzfileobj = np.load('out.npz') #returns NpzFile object
#print(type(npzfileobj))
print(npzfileobj.files)
print(npzfileobj['arr_0'])
print(npzfileobj['arr_1'])
```

```
D:\durgaclasses>py test.py
['arr_0', 'arr_1']
[[10 20 30]
 [40 50 60]]
[[ 70  80]
 [ 90 100]]
```

Note:
np.save() --->Save an array to a binary file in .npy format
np.savez()---->Save several arrays into a single file in .npz format but in uncompressed form.
np.savez_compressed()-->---->Save several arrays into a single file in .npz format but in compressed form.
np.load()--->To load/read arrays from .npy or .npz files.

compressed form:

```
import numpy as np
a = np.array([[10,20,30],[40,50,60]]) #2-D array with shape:(2,3)
b = np.array([[70,80],[90,100]]) #2-D array with shape:(2,2)

#save/serialize ndarrays object to a file
```

```
np.savez_compressed('out_compressed.npz',a,b)

#reading ndarray objects from a file
npzfileobj = np.load('out_compressed.npz') #returns NpzFile object
#print(type(npzfileobj))
print(npzfileobj.files)
print(npzfileobj['arr_0'])
print(npzfileobj['arr_1'])
```

Analysis:

```
D:\durgaclasses>dir out.npz  out_compressed.npz
 Volume in drive D has no label.
 Volume Serial Number is E2E9-F953

 Directory of D:\durgaclasses

29-Jun-21  11:00 AM             546 out.npz

 Directory of D:\durgaclasses

29-Jun-21  11:00 AM             419 out_compressed.npz
               2 File(s)            965 bytes
               0 Dir(s)  67,380,846,592 bytes free
```

Q. We can save object in compressed form, then what is the need of uncompressed form?

compressed form--->memory will be saved, but performance down.
uncompressed form--->memory won't be saved, but performance wise good.

Note:
if we are using save() function the file extension: npy
if we are using savez() or savez_compressed() functions the file extension: npz

Save ndarray objects to the file in normal text format:
---
savetxt() and loadtxt()

```
savetxt(fname, X, fmt='%.18e', delimiter=' ', newline='\n', header='', footer='',
comments='# ', encoding=None)
   Save an array to a text file.


loadtxt(fname, dtype=<class 'float'>, comments='#', delimiter=None,
converters=None, skiprows=0, usecols=None, unpack=False, ndmin=0,
encoding='bytes', max_rows=None, *, like=None)
   Load data from a text file.
```

```
import numpy as np
a = np.array([[10,20,30],[40,50,60]]) #2-D array with shape:(2,3)

#save/serialize ndarrays object to a file
np.savetxt('out.txt',a,fmt='%.1f')

#reading ndarray objects from a file and default dtype is float
out_array1 = np.loadtxt('out.txt')
print(out_array1)  `


#reading ndarray objects from a file and default dtype is int
out_array2 = np.loadtxt('out.txt',dtype=int)
```

```
print(out_array2)
```

```
TypeError: Mismatch between array dtype ('<U11') and format specifier ('%.18e
%.18e')
```

eg-2:

```
import numpy as np
a1 = np.array([['Sunny',1000],['Bunny',2000],['Chinny',3000],['Pinny',4000]])

#save this ndarray to a text file
np.savetxt('out.txt',a1,fmt='%s %s')

#reading ndarray from the text file
a2 = np.loadtxt('out.txt',dtype='str')
print(a2)
```

```
D:\durgaclasses>py test.py
[['Sunny' '1000']
 ['Bunny' '2000']
 ['Chinny' '3000']
 ['Pinny' '4000']]
```

out.txt:

```
-------
Sunny 1000
Bunny 2000
Chinny 3000
Pinny 4000
```

```
Zinny 5000
Vinny 6000
Minny 7000
Tinny 8000
```

creating ndarray from text file data:

```
import numpy as np
#reading ndarray from the text file
a2 = np.loadtxt('out.txt',dtype='str')
print(a2)
```

Writing ndarray objects to the csv file:
----
csv--->comma separated values

```
import numpy as np

a1 = np.array([[10,20,30],[40,50,60]])

#save/serialize to a csv file
np.savetxt('out.csv',a1,delimiter=',')

#reading ndarray object from a csv file
a2 = np.loadtxt('out.csv',delimiter=',')
print(a2)
```

Summary:
----
1. Save one ndarray object to the binary file(save() and load())
2. Save multiple ndarray objects to the binary file in uncompressed form(savez() and load())

3. Save multiple ndarray objects to the binary file in compressed form(savez_compressed() and load())
4. Save ndarry object to the text file (savetxt() and loadtxt())
5. Save ndarry object to the csv file (savetxt() and loadtxt() with delimiter=',')

Basic Statistics with Numpy:
----
100 years every minute temparature is available wrt hyderabad city.

Cricket Batsman
Data

In Datascience domain, we required to collect,store and analyze huge amount of data. From this data we may required to find some basic statistics like

1. Minimum value
2. Maximum value
3. Average Value
4. Sum of all values
5. Mean value
6. Median value
7. Variance
8. Standard deviation etc

1 met person--->IIIT Hyderabad-->DataScience
2019 elections in ap
politicians--->

10th+Intermediate+degree--->20 members
500 Rs List of questions

100 villages

20 samples
10 Lakhs from every mla candidate

running a shop

In Datascience domain, we required to collect,store and analyze huge amount of data. From this data we may required to find some basic statistics like

1. Minimum value
2. Maximum value
3. Average Value
4. Sum of all values
5. Mean value
6. Median value
7. Variance
8. Standard deviation etc

1. Minimum value:
----
np.min(a)
np.amin(a)
a.min()

amin(a, axis=None, out=None, keepdims=<no value>, initial=<no value>, where=<no value>)
    Return the minimum of an array or minimum along an axis.
    In n-dimensional arrays, if we are not providing axis, then it will be converted to 1-D array and returns minimum value.

eg-1: for 1-D array

```
>>> a = np.array([10,5,20,3,25])
>>> a
array([10,  5, 20,  3, 25])
>>> np.min(a)
3
>>> np.amin(a)
3
>>> a.min()
3

>>> a.amin()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'numpy.ndarray' object has no attribute 'amin'
```

eg-2: For 2-D array:

```
--------------------
a = np.array([[100,20,30],[10,50,60],[25,15,18],[4,5,19]])
>>> a
array([[100,  20,  30],
      [ 10,  50,  60],
      [ 25,  15,  18],
      [  4,   5,  19]])


>>> np.min(a)
4
>>> np.min(a)
4

>>> a
```

```
array([[100,  20,  30],
       [ 10,  50,  60],
       [ 25,  15,  18],
       [  4,   5,  19]])
>>> np.min(a,axis=0) #returns minimum row and that row contains 3 elements
array([ 4,  5, 18])
>>> np.min(a,axis=1) #returns minimum column and that column contains 4
elements
array([20, 10, 15,  4])
```

eg-3: for 2-D array:
----
a = np.arange(24).reshape(6,4)

```
>>> a = np.arange(24).reshape(6,4)
>>> a
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]])
>>> np.min(a)
0
>>> np.min(a,axis=0)
array([0, 1, 2, 3])
>>> np.min(a,axis=1)
array([ 0,  4,  8, 12, 16, 20])
```

eg-4:

```
>>> a = np.arange(24)
>>> a
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])
>>> np.random.shuffle(a)
>>> a
array([13, 19, 12, 15,  2,  6,  5, 14,  8, 21, 10, 11, 22,  0, 18,  4, 20,
      17,  7,  9, 23,  1,  3, 16])
>>> a = a.reshape(6,4)
>>> a
array([[13, 19, 12, 15],
      [ 2,  6,  5, 14],
      [ 8, 21, 10, 11],
      [22,  0, 18,  4],
      [20, 17,  7,  9],
      [23,  1,  3, 16]])
>>> np.min(a)
0
>>> np.min(a,axis=0)
array([2, 0, 3, 4])
>>> np.min(a,axis=1)
array([12,  2,  8,  0,  7,  1])
```

2. Finding Maximum value:

```
np.amax(a)
np.max(a)
a.max()
```

eg-1: For 1-D array:

```
>>> a = np.array([10,5,20,3,25])
>>> a
array([10,  5, 20,  3, 25])
>>> np.amax(a)
```

```
25
>>> np.max(a)
25
>>> a.max()
25
```

eg-2: for 2-D array:

```
>>> a = np.array([[100,20,30],[10,50,60],[25,15,18],[4,5,19]])
>>> a
array([[100,  20,  30],
       [ 10,  50,  60],
       [ 25,  15,  18],
       [  4,   5,  19]])
>>> np.max(a)
100
>>> np.max(a,axis=0)
array([100,  50,  60])
>>> np.max(a,axis=1)
array([100,  60,  25,  19])
```

3. Finding sum of elements:

```
np.sum(a)
a.sum()
```

```
>>> a = np.arange(4)
>>> a
array([0, 1, 2, 3])
>>> np.sum(a)
6
>>> a.sum()
6
```

```
>>> a = np.arange(9).reshape(3,3)
>>> a
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
>>> np.sum(a)
36
>>> a.sum()
36
>>> np.sum(a,axis=0)
array([ 9, 12, 15])
>>> np.sum(a,axis=1)
array([ 3, 12, 21])
```

Finding Mean value:

Mean is the sum of elements along the specified axis divided by number of elements.

sum of elements/number of elements

```
np.mean(a)
a.mean()
```

```
Help on function mean in module numpy:

mean(a, axis=None, dtype=None, out=None, keepdims=<no value>, *,
where=<no value>)
    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.

eg-1: for 1-D array:

```
---------------------
a = np.arange(5)


>>> a = np.arange(5)
>>> a
array([0, 1, 2, 3, 4])
>>> np.mean(a)
2.0
>>> a.mean()
2.0
```

eg-2: for 2-D array:

```
-------------------
>>> a = np.arange(9).reshape(3,3)
>>> a
array([[0, 1, 2],
      [3, 4, 5],
      [6, 7, 8]])
>>> np.mean(a)
4.0
>>> np.mean(a,axis=0)
array([3., 4., 5.])
>>> np.mean(a,axis=1)
array([1., 4., 7.])
>>> np.mean(a,dtype=int)
4
```

5. Finding Median value by using numpy.median() function:

Mean means average
Median means middle element of the array

10 20 30 --->20 (number of elements is odd)
10 20 30 40--->Average/Mean of middle 2 elements (number of elements is even)
--->25.0

np.median(a)

```
median(a, axis=None, out=None, overwrite_input=False, keepdims=False)
   Compute the median along the specified axis.
```

eg-1: for 1-D array:

```
>>> a = np.array([10,20,30,40])
>>> np.median(a)
25.0

>>> a = np.array([10,20,30])
>>> np.median(a)
20.0
```

eg-2: for 2-D array:

```
>>> a = np.arange(9).reshape(3,3)
>>> a
array([[0, 1, 2],
     [3, 4, 5],
     [6, 7, 8]])
>>> np.median(a)
```

```
4.0
>>> np.median(a,axis=0)
array([3., 4., 5.])
>>> np.median(a,axis=1)
array([1., 4., 7.])
```

eg-2: for 2-D array:

```
>>> a = np.arange(16).reshape(4,4)
>>> a
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15]])
>>> np.median(a)
7.5
>>> np.median(a,axis=0)
array([6., 7., 8., 9.])
>>> np.median(a,axis=1)
array([ 1.5,  5.5,  9.5, 13.5])
```

Note:
Mean means average where as Median means middle element

Finding Variance of the ndarray:

The variance is a measure of variability. It is calculated by taking the average of squared deviations from the mean.

average of
squared
deviations from the mean.

NUMPY contains var() function to find variance.

eg-1: For 1-D array:

----

a = [1,2,3,4,5]

mean(a) = 3.0

deviations from the mean: [-2.0,-1.0,0.0,1.0,2.0]

squares of deviations from the mean: [4.0,1.0,0.0,1.0,4.0]

Average of squares of deviations from the mean: 2.0===>VARIANCE

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

For 2-D array:

```
>>> a = np.arange(6).reshape(2,3)
>>> a
array([[0, 1, 2],
     [3, 4, 5]])
>>> np.var(a)
2.9166666666666665
>>> np.var(a,axis=0)
array([2.25, 2.25, 2.25])
>>> np.var(a,axis=1)
array([0.66666667, 0.66666667])
```

Note: ndarray class also contains var() method

```
np.var(a)
a.var()
```

Finding stadard deviation by using numpy.std() function:

Standard deviation is the square root of the variance.
Variance means the average of squares of deviations from the mean.

eg-1: for 1-D array:

```
>>> a = np.array([1,2,3,4,5])
>>> a
array([1, 2, 3, 4, 5])
>>> np.var(a)
2.0
>>> np.std(a)
1.4142135623730951
```

eg-2: For 2-D array:

```
>>> a = np.arange(9).reshape(3,3)
>>> a
array([[0, 1, 2],
      [3, 4, 5],
      [6, 7, 8]])
>>> np.var(a)
6.666666666666667
>>> np.std(a)
2.581988897471611
>>> np.var(a,axis=0)
array([6., 6., 6.])
>>> np.std(a,axis=0)
```

```
array([2.44948974, 2.44948974, 2.44948974])
>>> np.std(a,axis=1)
array([0.81649658, 0.81649658, 0.81649658])
```

Note: ndarray class also contains std() method

```
>>> a
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
>>> a.std()
2.581988897471611
>>> a.std(axis=0)
array([2.44948974, 2.44948974, 2.44948974])
>>> a.std(axis=1)
array([0.81649658, 0.81649658, 0.81649658])
```

Summary:

----

1. np.min(a)/np.amin(a)/a.min()--->Returns the minimum value of the array
2. np.max(a)/np.amax(a)/a.max()--->Returns the maximum value of the array
3. np.sum(a)/a.sum()--->Returns the Sum of values of the array
4. np.mean(a)/a.mean()--->Returns the arithmetic mean of the array.
5. np.median(a) --->Returns median value of the array
6. np.var(a)/a.var() --->Returns variance of the values in the array
7. np.std(a)/a.std() --->Returns Standard deviation of the values in the array

Numpy Mathematical Functions:

The functions which operates element by element on whole array, are called universal functions.

To perform mathematical operations numpy library contains several universal functions(ufunc).

1. np.exp(a) --->Takes e to the power of each value. e value: 2.7182
2. np.sqrt(a) --->Returns square root of each value.
3. np.log(a) --->Returns logarithm of each value.
4. np.sin(a) ---->Returns the sine of each value.
5. np.cos(a) --->Returns the co-sine of each value.
6. np.tan(a) --->Returns the tangent of each value.
etc

```
>>> a = np.array([[1,2],[3,4]])
>>> np.exp(a)
array([[ 2.71828183,  7.3890561 ],
       [20.08553692, 54.59815003]])
>>> a = np.arange(5)
>>> a
array([0, 1, 2, 3, 4])
>>> np.exp(a)
array([ 1.        ,  2.71828183,  7.3890561 , 20.08553692, 54.59815003])
>>> np.sqrt(a)
array([0.        , 1.        , 1.41421356, 1.73205081, 2.        ])
>>> np.log(a)
<stdin>:1: RuntimeWarning: divide by zero encountered in log
array([      -inf, 0.        , 0.69314718, 1.09861229, 1.38629436])
>>> np.sin(a)
array([ 0.        ,  0.84147098,  0.90929743,  0.14112001, -0.7568025 ])
>>> np.cos(a)
array([ 1.        ,  0.54030231, -0.41614684, -0.9899925 , -0.65364362])
>>> np.tan(a)
array([ 0.        ,  1.55740772, -2.18503986, -0.14254654,  1.15782128])
```

How to find unique items and count:

----

unique() function

```
unique(ar, return_index=False, return_inverse=False, return_counts=False,
axis=None)
   Find the unique elements of an array.
```

eg-1: To get array with unique values:

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

eg-2: To get indices also:

```
>>> items,indices = np.unique(a,return_index=True)
>>> items
array([1, 2, 3, 4, 5, 6])
>>> indices
array([ 0,  2,  3,  4, 13, 15], dtype=int64)
```

eg-3: To get count also:

```
>>> items,counts = np.unique(a,return_counts=True)
>>> a
array([1, 1, 2, 3, 4, 2, 3, 4, 4, 1, 2, 3, 4, 5, 5, 6])
>>> items
array([1, 2, 3, 4, 5, 6])
>>> counts
array([3, 3, 3, 4, 2, 1], dtype=int64)
```

eg-4: To get all:

```
>>> items,indices,counts = np.unique(a,return_index=True,return_counts=True)
>>> items
array([1, 2, 3, 4, 5, 6])
>>> indices
array([ 0,  2,  3,  4, 13, 15], dtype=int64)
>>> counts
array([3, 3, 3, 4, 2, 1], dtype=int64)
```

test.py:

```
import numpy as np
a = np.array(['a','a','b','c','a','a','b','c','a','b','d'])
items,indices,counts = np.unique(a,return_index=True,return_counts=True)
for item,index,count in
zip(np.nditer(items),np.nditer(indices),np.nditer(counts)):
        print(f"Element '{item}' occurred {count} times and its first occurrence
index:{index}")
```

```
D:\durgaclasses>py test.py
Element 'a' occurred 5 times and its first occurrence index:0
Element 'b' occurred 3 times and its first occurrence index:2
Element 'c' occurred 2 times and its first occurrence index:3
Element 'd' occurred 1 times and its first occurrence index:10
```

Creation of array by using diag() function:

--

```
diag(v, k=0)
   Extract a diagonal or construct a diagonal array.
```

```
Parameters
----------
    v : array_like
        If `v` is a 2-D array, return a copy of its `k`-th diagonal.
        If `v` is a 1-D array, return a 2-D array with `v` on the `k`-th
        diagonal.
    k : int, optional
        Diagonal in question. The default is 0. Use `k>0` for diagonals
        above the main diagonal, and `k<0` for diagonals below the main
        diagonal.
>>> a = np.arange(1,10).reshape(3,3)
>>> a
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
>>> np.diag(a,k=0)
array([1, 5, 9])
>>> np.diag(a,k=1)
array([2, 6])
>>> np.diag(a,k=1)
array([2, 6])
>>> np.diag(a,k=-1)
array([4, 8])
>>> np.diag(a,k=-2)
array([7])
>>> np.diag(a,k=-3)
array([], dtype=int32)


>>> a = np.array([10,20,30,40])
>>> np.diag(a,k=0)
array([[10,  0,  0,  0],
       [ 0, 20,  0,  0],
       [ 0,  0, 30,  0],
       [ 0,  0,  0, 40]])
>>> np.diag(a,k=1)
array([[ 0, 10,  0,  0,  0],
       [ 0,  0, 20,  0,  0],
       [ 0,  0,  0, 30,  0],
       [ 0,  0,  0,  0, 40],
       [ 0,  0,  0,  0,  0]])
>>> np.diag(a,k=-1)
array([[ 0,  0,  0,  0,  0],
       [10,  0,  0,  0,  0],
       [ 0, 20,  0,  0,  0],
       [ 0,  0, 30,  0,  0],
       [ 0,  0,  0, 40,  0]])
```

----

9. Creation of diagonal array by using diag() function:

--

Syntax:

diag(v, k=0)

-->Extract a diagonal or construct a diagonal array.

**-->If `v` is a 2-D array, return a copy of its `k`-th diagonal.
If `v` is a 1-D array, return a 2-D array with `v` on the `k`-th
diagonal.**

Creation of 2-D array with our provided diagonal elements:

--

eg-1: Creation of 2-D array with our provided diagonal elements

```
>>> a = np.diag([1,2,3])
>>> a
array([[1, 0, 0],
      [0, 2, 0],
      [0, 0, 3]])
eg-2:
>>> a = np.diag([1,2,3],k=1)
>>> a
array([[0, 1, 0, 0],
      [0, 0, 2, 0],
      [0, 0, 0, 3],
      [0, 0, 0, 0]])
eg-3:
>>> a = np.diag([1,2,3],k=-1)
>>> a
array([[0, 0, 0, 0],
      [1, 0, 0, 0],
      [0, 2, 0, 0],
      [0, 0, 3, 0]])
```

Extract diagonal elements from the given 2-D array:

```
>>> a = np.arange(16).reshape(4,4)
>>> a
array([[ 0,  1,  2,  3],
      [ 4,  5,  6,  7],
      [ 8,  9, 10, 11],
      [12, 13, 14, 15]])
eg-1:
>>> np.diag(a)
array([ 0,  5, 10, 15])
eg-2:
>>> np.diag(a,k=1)
array([ 1,  6, 11])
eg-3:
>>> np.diag(a,k=-1)
array([ 4,  9, 14])
```

View and Copy:

View:

View is not separate object and just it is logical representation of existing array. If we perform any changes to the original array, those changes will be reflected to the view. viceversa also.

We can create view explicitly by using view() method of ndarray class.

```
>>> a = np.array([10,20,30,40])
>>> b = a.view()
>>> b
array([10, 20, 30, 40])
>>> a
array([10, 20, 30, 40])
>>> a[0]=7777
>>> a
array([7777,   20,   30,   40])
>>> b
array([7777,   20,   30,   40])
```

Copy:

Copy means separate object.
If we perform any changes to the original array, those changes won't be reflected to the Copy. viceversa also.

By using copy() method of ndarray class, we can create copy of existing ndarray.

```
>>> a = np.array([10,20,30,40])
>>> b = a.copy()
>>> a
array([10, 20, 30, 40])
>>> b
array([10, 20, 30, 40])
>>> a[0]=7777
>>> a
array([7777,   20,   30,   40])
>>> b
array([10, 20, 30, 40])
```

==

Chapter-23: Numpy Practice Quesions Set-1

===

Q1. Create an array of 7 zeros?

```
>>> np.zeros(7)
array([0., 0., 0., 0., 0., 0., 0.])
>>> np.zeros(7,dtype=int)
array([0, 0, 0, 0, 0, 0, 0])
>>> np.full(7,0)
array([0, 0, 0, 0, 0, 0, 0])
```

Q2. Create an array of 9 ones?

```
>>> np.ones(9)
array([1., 1., 1., 1., 1., 1., 1., 1., 1.])
>>> np.ones(9,dtype=int)
array([1, 1, 1, 1, 1, 1, 1, 1, 1])
>>> np.full(9,1)
array([1, 1, 1, 1, 1, 1, 1, 1, 1])
```

Q3. Create an array of 10 fours?

```
>>> np.full(10,4)
array([4, 4, 4, 4, 4, 4, 4, 4, 4, 4])
>>> np.zeros(10,dtype=int)+4
array([4, 4, 4, 4, 4, 4, 4, 4, 4, 4])
>>> np.ones(10,dtype=int)+3
array([4, 4, 4, 4, 4, 4, 4, 4, 4, 4])
```

Q4. Create an array of integers from 10 to 40?

```
>>> np.arange(10,41)
array([10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26,
       27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40])
```

Q5. Create an array of integers from 10 to 40 which are even?

```
>>> np.arange(10,41,2)
array([10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40])
```

Q6. Create an array of integers from 10 to 40 which are odd?

```
>>> np.arange(11,41,2)
array([11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31, 33, 35, 37, 39])
```

Q7. Create an array of integers from 10 to 40 which are divisible by 7?

1st way:

```
-------
>>> np.arange(14,41,7)
array([14, 21, 28, 35])
```

2nd way:

```
--------
>>> a = np.arange(10,41)
>>> a
array([10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26,
       27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40])
>>> a[a%7==0]
array([14, 21, 28, 35])
```

Q8. Create a numpy array having 10 numbers starts from 24 but only even numbers?

1st way:

```
-------
>>> np.arange(24,43,2)
array([24, 26, 28, 30, 32, 34, 36, 38, 40, 42])
```

2nd way:

```
-------
>>> a = np.arange(24,50)
>>> a
array([24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40,
       41, 42, 43, 44, 45, 46, 47, 48, 49])
>>> a = a[a%2==0]
>>> a
array([24, 26, 28, 30, 32, 34, 36, 38, 40, 42, 44, 46, 48])
>>> np.resize(a,10)
array([24, 26, 28, 30, 32, 34, 36, 38, 40, 42])
```

Q9. Create a 4X4 matrix with elements from 1 to 16?

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

Q10. Create a 4X4 identity matrix?

```
>>> np.eye(4)
array([[1., 0., 0., 0.],
      [0., 1., 0., 0.],
      [0., 0., 1., 0.],
      [0., 0., 0., 1.]])
>>> np.identity(4)
array([[1., 0., 0., 0.],
      [0., 1., 0., 0.],
      [0., 0., 1., 0.],
      [0., 0., 0., 1.]])
```

Q11. By using numpy module, generate ndarray with random numbers from 1 to 100 with the shape:(2,3,4)?

```
>>> np.random.randint(1,101,size=(2,3,4))
array([[[74, 40, 32, 67],
       [93, 34, 31, 86],
       [ 4,  3, 54, 90]],

      [[11, 25, 49, 38],
       [ 8, 20, 40, 59],
       [60, 88, 83, 51]]])
```

Q12. By using numpy module, generate a random number between 0 and 1?

```
>>> np.random.rand()
0.6563606029950465
>>> np.random.rand()
0.33598671990249174
>>> np.random.rand()
0.15195445013829945
>>> np.random.rand()
0.5626155619658889
>>> np.random.rand()
0.6960796589387932
>>> np.random.rand()
0.6192875505685667
>>> np.random.rand()
0.16912615729913438
```

Q13. By using numpy module, generate an array of 10 random samples from a uniform distribution over [0,1)?

```
np.random.rand(10)
>>> np.random.rand(10)
array([0.23175153, 0.23516775, 0.16853863, 0.4361167 , 0.43694742,
       0.24545343, 0.974236  , 0.64757367, 0.0890843 , 0.3444159 ])
```

Q14. By using numpy module, generate an array of 10 random samples from a uniform distribution over [10,20)?

```
np.random.uniform(low=0.0,high=1.0,size=None)

>>> np.random.uniform(10,20,10)
```

```
array([19.80499139, 11.35811947, 11.31370507, 14.94415343, 17.25710869,
       12.40993842, 16.25033344, 17.10067103, 10.18653984, 19.31369384])
```

Q15. By using numpy module, generate an array of 10 random samples from a standard normal distribution of mean 0 and standard deviation 1?

```
np.random.randn(10)
```

Q16. By using numpy module, generate an array of 10 random samples from a standard normal distribution of mean 15 and and stadard deviation 4?

```
normal(loc=0.0, scale=1.0, size=None)
loc--->mean
scale--->standard deviation

a = np.random.normal(15,4,10)
```

Q17. Create an array of 10 linearly spaced points between 1 and 100?

```
>>> np.linspace(1,100,10)
array([  1.,  12.,  23.,  34.,  45.,  56.,  67.,  78.,  89., 100.])
```

Q18. Create an array of 15 linearly spaced points between 0 and 1?

```
>>> np.linspace(0,1,15)
array([0.        , 0.07142857, 0.14285714, 0.21428571, 0.28571429,
       0.35714286, 0.42857143, 0.5       , 0.57142857, 0.64285714,
      0.71428571, 0.78571429, 0.85714286, 0.92857143, 1.        ])
```

Chapter-24: Numpy Practice Quesions Set-2

```
==========================================
>>> a = np.arange(1,37).reshape(6,6)
>>> a
array([[ 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, 30],
       [31, 32, 33, 34, 35, 36]])
```

Diagram-27: diagram_27

Q1. How to access element 16?

```
>>> a[2][3]
16
>>> a[2,3]
16
```

Q2. To get the following array:

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

```
>>> a[0:1,1:5]
array([[2, 3, 4, 5]])
```

Q3. To get the following array:

```
array([2, 3, 4, 5])
>>> a[0,1:5]
array([2, 3, 4, 5])
```

Q4. To get the following array

```
array([[ 1,  2,  3,  4,  5,  6],
      [31, 32, 33, 34, 35, 36]])
```

```
>>> a[::5,:]
array([[ 1,  2,  3,  4,  5,  6],
      [31, 32, 33, 34, 35, 36]])
```

Q5. To get the following array

```
array([[ 9, 10],
      [15, 16],
      [21, 22]])
```

```
>>> a[1:4,2:4]
array([[ 9, 10],
      [15, 16],
      [21, 22]])
```

Q6. Create 1-D array with all even numbers of a?

```
>>> a[a%2==0]
array([ 2,  4,  6,  8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34,
       36])
```

Q7. Create a 1-D array with elements of a which are divisible by 5?

```
>>> a[a%5==0]
array([ 5, 10, 15, 20, 25, 30, 35])
```

Q8. Create 1-D array with elements 8,17,26 and 35. We have to use elements of a?

```
>>> a[[1,2,4,5],[1,4,1,4]]
array([ 8, 17, 26, 35])
```

Q9. Select minimum element of this ndarray?

```
np.amin(a)
np.min(a)
a.min()
```

```
>>> np.amin(a)
1
>>> np.min(a)
1
>>> a.min()
1
```

Q10. Select maximum element of this ndarray?

```
np.amax(a)
np.max(a)
a.max()

>>> np.amax(a)
36
>>> np.max(a)
36
>>> a.max()
36
```

Q11. Find sum of elements present inside this array?

```
>>> np.sum(a)
666
>>> a.sum()
666
```

Q12. Find sum of elements along axis-0?

```
np.sum(a,axis=0)

>>> np.sum(a,axis=0)
array([ 96, 102, 108, 114, 120, 126])
```

Q13. Find mean of this array?

```
>>> np.mean(a)
18.5
```

Q14. Find median of this array?

```
>>> np.median(a)
18.5
```

Q15. Find variance of this array?

```
>>> np.var(a)
107.91666666666667
```

Q16. Find standard deviation of this array?

```
>>> np.std(a)
10.388294694831615
```

Chapter-25: Numpy Quiz Questions

==================================

Q1. Consider the following code:

```
import numpy as np
a = np.array([])
print(a.shape)
```

What is the output ?

A. (0,)
B. (1,)
C. (1,1)
D. 0

Ans: A

Q2. Consider the following code:

```
import numpy as np
a = np.arange(10,20,-1)
b = a.reshape(5,2)
print(b.flatten())
```

What is the output?

A. [10 11 12 13 14 15 16 17 18 19]
B. [20 19 18 17 16 15 14 13 12 11]
C. array([])
D. ValueError

Ans: D

Q3. Consider the following code:

```
import numpy as np
a = np.arange(1,6)
a = a[::-2]
print(a)
```

What is the output ?

A. [1 2 3 4 5]
B. [6 4 2]
C. [5 3 1]
D. [4 2]

Ans: C

Q4. Consider the following code:

```
a = np.array([3,3])
b = np.array([3,3.5])
c = np.array([3,'3'])
d = np.array([3,True])
```

The dtypes of a,b,c and d are:

A. int,float,str,int
B. int,int,str,bool
C. float,float,int,int
D. int,float,str,bool
E. ValueError while creating b,c,d

Ans: A

Q5. Consider the code:
a = np.array([1,2,3,2,3,4,5,2,3,4,1,2,3,6,7])
print(a[2:5])

What is the output?

A. [3 2 3]
B. [2 3 2]
C. [3 2 3 4]
D. IndexErrror

Ans: A

Q6. Consider the code:
a = np.array([1,2,3,2,3,4,5,2,3,4,1,2,3,6,7])
print(a[2:7:2])

What is the output?

A. [3 3 5]
B. [3 2 3 4 5]
C. [3 2 3 4]
D. IndexErrror

Ans: A

Q7. Consider the code:

```
a = np.array([1,2,3,2,3,4,5,2,3,4,1,2,3,6,7])
print(a[:3:3])
```

What is the output?

A. [1]
B. [1 2 3]
C. [1 3]
D. IndexErrror

Ans: A

Q8. Consider the code:

```
a = np.array([1,2,3,2,3,4,5,2,3,4,1,2,3,6,7])
print(a[7:2:2])
```

What is the output?

A. [2 4 2]
B. [2 5 4 3]
C. []
D. IndexErrror

Ans: C
Q9. Consider the code:

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

What is the output?

A. [1 2 4]

B. [2 3 3]
C. [True True True]
D. IndexErrror

Ans: B
Q10. Consider the ndarray:

a = np.arange(20).reshape(5,4)

Which of the following options will provide 15?

A. a[3][3]
B. a[-2][-1]
C. a[3][-1]
D. a[3,3]
E. All of these

Ans: E

Q11. We can create Numpy array only to represent 2 Dimensional matrix?

A. True
B. False

Ans: B

Q12. Which of the following is valid way of importing numpy library?

A. from numpy import np
B. import numpy as np
C. import numpy as np1
D. import np as numpy

Ans: B and C

Q13. Which of the following cannot be used as numpy index?

A. 0
B. 1
C. -1
D. -2
E. None of these

Ans: E

Q14. Which of the following is correct syntax to create a numpy array?

A. np.array([10,20,30,40])
B. np.createArray([10,20,30,40])
C. np.makeArray([10,20,30,40])
D. np([10,20,30,40])

Ans: A

Q15. Which of the following are valid ways of finding the number of dimensions of input array a?

A. np.ndim(a)
B. np.dim(a)
C. a.ndim()
D. a.ndim

Ans: A and D

Q16. Consider the array
a = np.array([10,20,30,40])

Which of the following prints first element of this array?

A. print(a[0])
B. print(a[1])
C. print(a.0)
D. print(a.1)

Ans: A

Q17. How to check data type of ndarray a?

A. np.dtype(a)
B. a.dtype()
C. a.dtype
D. All of these

Ans: C

Q18. Which of the following is valid of creating float type ndarray?

A. a = np.array([10,20,30,40],dtype='f')
B. a = np.array([10,20,30,40],dtype='float')
C. a = np.array([10,20,30,40],dtype=float)
D. a = np.array([10,20,30,40])

Ans: A,B,C

Q19. Which of the following statements are valid?

A. If we perform any changes to the original array, then those changes will be reflected to the VIEW.

B. If we perform any changes to the original array, then those changes won't be reflected to the VIEW.
C. If we perform any changes to the original array, then those changes will be reflected to the COPY.
D. If we perform any changes to the original array, then those changes won't be reflected to the COPY.

Ans: A,D

Q20. Choose only one appropriate statement regarding shape of ndarray?

A. The shape represents the size of each dimension.
B. The shape represents only the number of rows.
C. The shape represents only the number of columns.
D. The shape represents the total size of the array.

Ans: A
(2,3,4)
a.size

Q21. By using which functions of numpy library, we can perform search operation for the required elements?

A. find()
B. search()
C. where()
D. All of these

Ans: C

Q22. Consider the following array:
a = np.array([10,20,30,10,20,10,10,30,40])

Which of the following code represents the indices where element 10 present?

A. np.find(a == 10)
B. np.where(a == 10)
C. np.search(a == 10)
D. None of these

Ans: B

Q23. Which of the following code collects samples from uniform distribution of 1000 values in the interval [10,100)?

A. np.uniform(10,100,size=1000)
B. np.random.uniform(10,100,size=1000)
C. np.random.uniform(low=10,high=100,size=1000)
D. np.random.uniform(from=10,to=100,size=1000)

Ans: B,C

np.random.rand()--->uniform distribution over [0,1)
np.random.uniform()--->uniform distribution over [a,b)

uniform(low=0.0, high=1.0, size=None)

Q24. Which of the following code collects samples from normal distribution of 1000 values with the mean 10 and standard deviation 0.3?

A. np.normal(10,0.3,1000)
B. np.random.normal(10,0.3,1000)
C. np.random.normal(mean=10,std=0.3,size=1000)

D. np.random.normal(loc=10,scale=0.3,size=1000)

Ans: B and D
normal(loc=0.0, scale=1.0, size=None)

loc : float or array_like of floats
    Mean ("centre") of the distribution.
  scale : float or array_like of floats
    Standard deviation (spread or "width") of the distribution. Must be non-negative.

Q25. Which of the following is valid of performing add operation at element level of two ndarrays a and b?

A. np.add(a,b)
B. np.sum(a,b)
C. np.append(a,b)
D. a+b

Ans: A,D

Q26. Which of the following is valid of performing subtract operation at element level of two ndarrays a and b?

A. a-b
B. np.minus(a,b)
C. np.min(a,b)
D. np.subtract(a,b)

Ans: A,D

Q27. Which of the following returns 1.0 if a = 1.2345?

A. np.trunc(a)
B. np.fix(a)
C. np.around(a)
D. All of these

Ans: D

np.trunc(a):

Remove the digits after decimal point

```
>>> np.trunc(1.23456)
1.0
>>> np.trunc(1.99999)
1.0
```

np.fix(a):

Round to nearest integer towards zero

```
>>> np.fix(1.234546)
1.0
>>> np.fix(1.99999999)
1.0
```

np.around(a):

It will perform round operation.
If the next digit is >=5 then remove that digit by incrementing previous digit.
If the next digit is <5 then remove that digit and we are not required to do anything with the previous digit.

```
>>> np.around(1.23456)
```

```
1.0
>>> np.around(1.99999)
2.0
```

Q28. Which of the following returns 2.0 if a = 1.995?

A. np.trunc(a)
B. np.fix(a)
C. np.around(a)
D. All of these

Ans: C

Q29. Which of the following are valid ways of creating a 2-D array?

A. np.array([[10,20,30],[40,50,60]])
B. np.array([10,20,30,40,50,60])
C. np.array([10,20,30,40,50,60],ndim=2)
D. np.array([10,20,30,40,50,60],ndmin=2)

Ans: A,D

Q30. Consider the code:

```
a = np.array([10,20,30,40])
print(np.cumsum(a))
```

What is the result?

A. [10 20 30 40]
B. [10 30 60 100]
C. [100 100 100 100]

D. None of these

Ans: B

Q31. Consider the array:
a = np.array([10, 15, 20, 25, 30, 35, 40])
Which of the following selects items from the second item to fourth item?

A. a[1:4]
B. a[1:5]
C. a[2:5]
D. a[2:4]

Ans: A

Q32. Consider the array:
a = np.array([10, 15, 20, 25, 30, 35, 40])
Which of the following selects items from the third item to fourth item?

A. a[1:4]
B. a[1:5]
C. a[2:5]
D. a[2:4]

Ans: D

Q33. Consider the array:
a = np.array([10, 15, 20, 25, 30, 35, 40])
Which of the following selects every other item from the second item to sixth item?

A. a[1:6:2]
B. a[1:5:2]
C. a[2:6:2]
D. a[2:7:2]

Ans: A

Q34. Consider the array:
a = np.array([10, 15, 20, 25, 30, 35, 40])
Which of the following selects every other item from total array?

A. a[:7:2]
B. a[0::2]
C. a[0:7:2]
D. a[::2]
E. All of these

Ans: E

Q35. Consider the following array:
a = np.array([10.5,20.6,30.7])

Which of the following is the valid way to convert array into int data type?

A. newarray = a.int()
B. newarray = a.asInt()
C. newarray = a.astype(int)
D. newarray = a.astype('int')
E. newarray = np.int32(a)

Ans: C,D,E

Q36. Consider the following array:
a = np.array([[10,20,30],[40,50,60]])
Which of the following is valid way to get element 50?

- **A. a[1][1]**
- **B. a[-1][-2]**
- **C. a[1][-2]**
- **D. a[-1][1]**
- **E. All the above**

Ans: E

Q37. Consider the following array:
a = np.array([[10,20,30],[40,50,60]])
Which of the following is valid way to get element 30?

- **A. a[0][2]**
- **B. a[-2][-1]**
- **C. a[0][-1]**
- **D. a[-2][2]**
- **E. All the above**

Ans: E

Q38. Consider the following array:
a =
np.array([[[1,2,3,4],[5,6,7,8],[9,10,11,12]],[[13,14,15,16],[17,18,19,20],[21,22,23,24]]])
Which of the following is valid way to get element 18?

- **A. a[1][1][1]**
- **B. a[-1][-2][-3]**
- **C. a[1][-2][-3]**
- **D. a[1][1][-3]**
- **E. a[-1][1][1]**
- **F. a[-1][-2][1]**

G. All the above

Ans: G

Q39. Consider the following array:

```
a =
np.array([[[1,2,3,4],[5,6,7,8],[9,10,11,12]],[[13,14,15,16],[17,18,19,20],[21,22,23,
24]]])
```

Which of the following is valid way to get element 7?

A. a[0][1][2]
B. a[-2][-2][-2]
C. a[0][1][3]
D. a[-2][-2][2]

Ans: A,B,D

https://www.w3schools.com/python/numpy/numpy_quiz.asp

----

t.me/durgasoftupdates

Chapter-26: Numpy Interview Questions

=======================================

Q1. What is NumPy?

Numpy is Python based library which defines several functions to create and manage arrays and to perform complex mathematical operations in Datascience domain.

1. NumPy stands for Numerical Python Library.
2. Numpy library defines several functions to solve complex mathematical problems in Data Science, Machine Learning, Deep Learning etc
3. Numpy acts as Backbone for most of the libraries used in Data Science like Pandas , sklearn etc

4. Numpy is developed on top of Numeric Library in 2005.
Numeric Library developed by Jim Hugunin.
Numpy is developed by Travis Oliphant and multiple contributors.
5. It is open source library and freeware.
6. The Fundamental data structure in numpy is ndarray.
ndarray --->N-Dimensional Array or Numpy Array
7. Numpy is written in C and Python Languages.
8. Numpy is superfast when compared with traditional python code.

Q2. What are the uses of NumPy?

Numpy Library defines several functions

1. For Creation and manipulation of multi dimensional arrays, which is the most commonly used data structure in the datascience domain.
2. For Mathematical operations includes trigonometric operations, statistical operations and algebraic computations.
3. Solving Differential equations

Q3. Why is NumPy preferred to other programming tools such as Idl, Matlab, Octave, Or Yorick?

Numpy is opensource and freeware.
Numpy is a high performance library as it is developed in c and python.
Easy to use as it is written in python language.
It defines several easy to use functions for mathematical operations like trigonometric operations, statistical operations and algebraic computations.

Q4. What are the various features of NumPy?

1. It is opensource and freeware.
2. It is superfast because it is developed in C language.

3. Numpy acts as backbone for Data Science Libraries like pandas, scikit-learn etc
  Pandas internally used 'nd array' to store data, which is numpy data structure.
  Scikit-learn internally used numpy's nd array.
4. Numpy has vectorization operations which can be performed at element level.
5. It defines several easy to use functions for mathematical operations like trigonometric operations, statistical operations and algebraic computations.

Question 5: How to Install NumPy

2 ways

1st way:

By using Anaconda Distribution
  Anaconda is python flavour for Data Science,ML etc.
  Anaconda distribution has inbuilt numpy library and hence we are not required to install.

2nd way:

If Python is already installed in our system, then we can install numpy library as follows

```
pip install numpy
```

```
D:\durgaclasses>pip install numpy
Collecting numpy
  Downloading numpy-1.20.2-cp38-cp38-win_amd64.whl (13.7 MB)
     |████████████████████████████████| 13.7 MB 6.4 MB/s
Installing collected packages: numpy
```

Successfully installed numpy-1.20.2

How to check installation:

```
-------------------------
D:\durgaclasses>py
Python 3.8.6 (tags/v3.8.6:db45529, Sep 23 2020, 15:52:53) [MSC v.1927 64 bit
(AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> import numpy
>>> numpy.__version__
'1.20.2'
```

Q6. What are various similarities between NumPy Arrays and Python lists?

1. Both can be used to store data.
2. The order will be preserved in both types. Hence we can access elements by using index.
3. Slicing is also applicable for both.
4. Both are mutable, ie once we create list or array, we can change its elements.

Q7. What are various differences between NumPy Arrays and Python lists?

1. List is inbuilt data type but numpy array is not inbuilt. To use numpy arrays, we have to install and import numpy library explicitly.

2. List can hold heterogeneous (Different types) elements.
eg: l = [10,10.5,True,'durga']

But array can hold only homogeneous elements.
eg: a = numpy.array([10,20,30])

3. On arrays we can perform vector operations(the operations which can be operated on every element of the array). But we cannot perform vector operations on list.

```
>>> import numpy as np
>>> a = np.array([10,20,30])
>>> a
array([10, 20, 30])
>>> a+1
array([11, 21, 31])
>>> a*2
array([20, 40, 60])
>>> a/2
array([ 5., 10., 15.])
>>> l=[10,20,30]
>>> l+1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate list (not "int") to list
>>> l*2
[10, 20, 30, 10, 20, 30]
>>> l/2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for /: 'list' and 'int'
```

4. Arrays consume less memory when compared with list.

eg:

```
import numpy
import sys
a = numpy.array([10,20,30,40,10,20,30,40,10,20,30,40,10,20,30,40])
l = [10,20,30,40,10,20,30,40,10,20,30,40,10,20,30,40]
print('The Size of Numpy Array:', sys.getsizeof(a))
print('The Size of List:', sys.getsizeof(l))
```

```
D:\durgaclasses>py test.py
The Size of Numpy Array: 168
The Size of List: 184
```

5. Arrays are Super Fast when compared with list.
6. Numpy Arrays are more convenient to use while performing mathematical operations.

Q8. What are the advantages NumPy Arrays over Python lists?

1. Performance wise Arrays are Super Fast when compared with list.
2. Arrays consume less memory when compared with list.
3. On arrays we can perform vector operations(the operations which can be operated on every element of the array). But we cannot perform vector operations on list.

Q9. What are the advantages Python Lists over NumPy Arrays?

1. List is inbuilt data type but numpy array is not inbuilt. To use numpy arrays, we have to install and import numpy library explicitly.
2. List can hold heterogeneous (Different types) elements.
```
  eg: l = [10,10.5,True,'durga']
```
But array can hold only homogeneous elements.
```
  eg: a = numpy.array([10,20,30])
```

Q10. How to create 1-D array and 2-D array from python lists?

```
1d_arrray = np.array([10,20,30,40])
2d_arrray = np.array([[10,20],[30,40]])
```

Q11. How to create a 3D array from Python Lists?
3d_arrray = np.array([[[10,20],[30,40]],[[50,60],[70,80]]])

Q12. How to find shape of nd array?
By using shape variable.

```
a = np.array([[[10,20],[30,40]],[[50,60],[70,80]]])
print(a.shape) #(2,2,2)
```

Q13. How to find data type of ndarray?
By using dtype variable.

```
a = np.array([[[10,20],[30,40]],[[50,60],[70,80]]])
print(a.dtype) #int32
```

Q14. How to count the number of times a given value appears in an array of integers?

We have to use unique() function of numpy library.

```
>>> a = np.array([1,1,2,3,4,2,3,4,1,2,3,4,5,5,6])
>>> items,count = np.unique(a,return_counts=True)
>>> items
array([1, 2, 3, 4, 5, 6])
>>> count
array([3, 3, 3, 3, 2, 1], dtype=int64)
```

Q15. How to check whether array is empty or not i.e array contains zero number of elements or not?

By using size variable.

```
>>> a = np.array([10,20,30])
>>> a.size
```

```
3
>>> a = np.array([])
>>> a.size
0
```

Q16. How to find the indices of an array in NumPy where some condition is true?
By using where() function.

```
>>> a = np.array([20,25,30,35,40])
>>> a
array([20, 25, 30, 35, 40])
>>> np.where(a%10==0)
(array([0, 2, 4], dtype=int64),)
```

Note: To get elements satisfy a condition:

```
>>> a[a%10==0]
array([20, 30, 40])
```

Q17. Given a 4x4 Numpy matrix, how to reverse the matrix?

1st way:

```
a = np.arange(16).reshape(4,4)
a.flatten()[::-1].reshape(4,4)
```

2nd way:

Numpy Library contains flip() function

```
a = np.arange(16).reshape(4,4)
>>> np.flip(a)
array([[15, 14, 13, 12],
[11, 10,  9,  8],
[ 7,  6,  5,  4],
[ 3,  2,  1,  0]]])
```

Q17. Create a 3x3x3 array with random values

```
#a= np.random.rand(3,3,3)
a= np.random.random((3,3,3))
print(a)
```

Q18. Create a 10x10 array with random values and find the minimum and maximum values

```
a = np.random.rand(10,10)
#a = np.random.random((10,10))
amin, amax = a.min(), a.max()
print(amin, amax)
```

Q19. Create a random vector of size 30 and find the mean value

```
a = np.random.random(30)
#a = np.random.rand(30)
b = a.mean()
print(b)
```

Q20. Create a 2d array of shape (10,10) with 1 on the border and 0 inside

```
a = np.ones((10,10),dtype=int)
#a[1:-1,1:-1] = 0
a[1:9,1:9] = 0
print(a)
```

o/p:

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

Q21. By using numpy, create the following array?

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

```
>>> a = np.ones((10,10),dtype=int)
>>> a[1:9,1:9] = 0
>>> a[[1,2,3,4,5,6,7,8],[1,2,3,4,5,6,7,8]]=1
>>> print(a)
```

Q22. By using numpy, create the following array?

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

```
a = np.ones((10,10),dtype=int)
a[1:9,1:9]=2
a[2:8,2:8]=3
a[3:7,3:7]=4
a[4:6,4:6]=5
```

Q23. How to find common values between two arrays?

```
 a = np.random.randint(0,10,10)
 b = np.random.randint(0,10,10)
 print(np.intersect1d(a,b))
```

o/p:

```
[0 7 9]
```

Q24. Create a random vector of size 10 and sort it

```
   a = np.random.random(10)
   a.sort()
   print(a)
```

Q25. Consider the following code:

```
 a = np.arange(9).reshape(3,3)
 for index, value in np.ndenumerate(a):
        print(index, value)
```

Write equivalent code without using ndenumerate() function?

Sol:

```
for index in np.ndindex(a.shape):
        print(index, a[index])
```

o/p:

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

Q26. How to access multiple elements of array which are not in order?

If we want to access multiple elements which are not in order(arbitrary elements) then we cannot use basic indexing and slicing operators. For this requirement we should go for Advanced Indexing.

Syntax: a[[row_indices],[column_indices]]

Q27. Consider the array:
a = np.array([10,20,30,40,50,60,70,80,90])
How to access the elements of 10,30,80?

1st way:

```
>>> indexes = np.array([0,2,7])
>>> a[indexes]
    array([10, 30, 80])
```

2nd way:

```
>>> l = [0,2,7]
 >>> a[l]
array([10, 30, 80])
>>> a[[0,2,7]]
array([10, 30, 80])
```

Q28. What are various differences between Slicing and Advanced Indexing?

1. The elements should be ordered and we cannot select arbitrary elements.
1. The elements need not be ordered and we can select arbitrary elements.
2. Condition based selection not possible.
2. Condition based selection is possible.
3. In numpy slicing, we wont get a new object just we will get view of the original object. If we perform any changes to the original copy, those changes will be reflected to the sliced copy.
3. But in the case of advanced indexing, a new separate copy will be created. If we perform any changes in one copy, then those changes won't be reflected in other.

Q29. what is broadcasting?

Even though, arrays are of different dimensions, different shapes and different sizes, still we can perform arithmetic operations between them. It is possible by broadcating.

Broadcasting will be performed automatically and we are not required to perform explicitly. But being developer, we should aware, when broadcasting will be performed and how it will be performed.

Broadcasting won't be possible in all cases. It has some rules. If the rules are satisfied then only broadcasting will be performed internally while performing arithmetic operations.

Q30. In how many ways we can iterate elements of ndarray?

There are three ways of iterate element of nd array:

1. **By using python's loops**
2. **By using numpy nditer() function**
3. **By using numpy ndenumerate() function**

Q31. Consider the array:
a = np.array([[[10,20],[30,40]],[[40,50],[60,70]]])
How to get one by one value using for loop?

```
import numpy as np
a = np.array([[[10,20],[30,40]],[[40,50],[60,70]]])
print(a)
print('Elements one by one:')
for x in a: # x is 2-D array
  for y in x: #y is 1-D array
    for z in y: #z is scalar
           print(z)
```

Q32. What is the purpose of nditer() function?

nditer() function is specially designed function to iterate elements of any n-D array easily without using multiple loops.

Q33. How to get required data type elements while iterating by using nditer() function?

While iterating elements of nd array, we can specify our required type. For this, we have to use op_dtypes argument.

Q34. Explain differences between normal for loop and nditer() function?

1. To iterate n-d array, n for loops are required.
1. To iterate n-d array, only one for loop is required.
2. There is no way to specify our required type
2. There is a way to specify our required type (op_dtypes argument)

Q35. What is the purpose of ndenumerate() function?

By using nditer() we will get elements only but not indexes.

If we want indexes also inaddition to elements, then we should use ndenumerate() function.

ndenumerate() function returns multidimensional index iterator which yields pairs of array indexes(coordinates) and values.

Q36. what is the purpose of reshape() function and it syntax?

We can use reshape() function to change array shape without changing data.

Syntax:

```
reshape(a, newshape, order='C')
```

Q37. What is the meaning of -1 argument in reshape() function?

If we don't know the size of any dimension, we can use -1 so that numpy itself will evaluate that size.

Q38. What is the purpose of resize() function and its syntax?

We can use resize() function to resize an existing array.

The new array is larger than the original array, then the new array is filled with repeated copies of 'a'. Note that this behavior is different from a.resize(new_shape) which fills with zeros instead of repeated copies of 'a'.

Syntax:

numpy.resize(a, new_shape)-->For extra elements repeated copies of a will be reused.
a.resize(new_shape) which fills with zeros instead of repeated copies of 'a'.

Q39. Differences between numpy.resize() and ndarray.resize() functions?

1. It is library function present in numpy module
1. It is method present in ndarray class.

2. It will creates a new array and returns it.
2. It won't return new array and existing array will be modified.

3. If the new_shape requires more elements then repeated copies of original array will be used.
3. If the new_shape requires more elements then extra elements filled with zeros.

Q40. Differences between reshape() and resize() functions?

1. It won't create new array object and just we will get view of existing array.
If we perform any changes in the reshaped copy, automatically those changes will be reflected in original copy.
1. It will create new array object with required new shape.
If we perform any changes in the resized array, those changes won't be reflected in original copy.

2. The reshape will be happend without changing original data.
2. There may be a chance of data change in resize()

3. The sizes must be matched.
3. The sizes need not be matched.

4. In unknown dimension we can use -1.

4. -1, such type of story not applicable.

Q41. What is the purpose of flatten() function and its syntax?
We can use flatten() method to flatten(convert) any n-Dimensional array to 1-Dimensional array.
Syntax:
```
ndarray.flatten(order='C')
```

Q42. what is the purpose of ravel() function?
We can use ravel() to flatten any n-Dimensional array to 1-D array, But it will return view but not separate copy.

Q43. what is the purpose of swapaxes and its syntax?
By using transpose we can interchage any number of dimensions. But if we want to interchange only two dimensions then we should go for swapaxes() function. It is the special case of transpose() function.

Syntax:
```
numpy.swapaxes(a, axis1, axis2)
```

Q44. which function we can use for searching the element of ndarray?
We can search elements of ndarray by using where() function.

Q45. How to insert elements into ndarrays?
We can insert elements into ndarrys by using the following functions.
1. insert()
2. append()

Q46. what is dot() function?
If we want to do matrix multiplication we should go for dot() function.

In Linear algebra, multiplication can be represented by using dot(.). Hence the name dot function.

eg: A.B

Q47. What is mean?

Mean is the sum of elements along the specified axis divided by number of elements.

Q48. What is numpy.median?

Median is middle element of the array.

If the total number of elements is even then two middle elements will be there. In this case median is average/mean of middle 2 elements.

Q49. What is varience?

The variance is a measure of variability. It is calculated by taking the average of squared deviations from the mean.

ie, variance = mean(abs(x-x.mean())**2)

Q50.What is Standard Deviation?

Standard deviation is the square root of the average of square deviations from mean.

The formula is std = sqrt(mean(abs(x-x.mean())**2))

Q51. list out some numpy universal functions(ufunc)?

1. np.exp(a) --->Takes e to the power of each value. e value: 2.718281828459045
2. np.sqrt(a) --->Returns square root of each value.
3. np.log(a) -->Returns logarithm of each value.
4. np.sin(a) --->Returns the sine of each value.
5. np.cos(a) --->Returns the co-sine of each value.
6. np.tan(a) --->Returns the tangent of each value.

Q52. By using which functions we can perform splitting ndarray?

1. split()
1. vsplit()
2. hsplit()
3. dsplit()
5. array_split()

Q53. List out some functions in the numpy random library?

1. randint()
2. rand()
3. uniform()
4. randn()
5. normal()
6. shuffle()

Q54. What is the difference between insert() and append() functions?

By using insert() function, we can insert elements at our required index position.

If we want to add elements always at end of the ndarray, then we have to go for append() function.

Numpy
Matplotlib
Pandas
Seaborn
plotly
....

Matplotlib:

Numpy --->Data Analysis Library
Pandas--->Data Analysis Library/Visualization library
Matplotlib/Seaborn/Plotly --->Data Visualization Libraries

1.1 Need of data visualization:
----
text form
graphical form

Data visualization is the representation of data in visual format.
Advantages:

1. **We can compare very easily.**
2. **We can identify relationships very easily.**
3. **We can identity symmetry and patterns between data.**
4. **We can analyze very easily.**

etc

There are multiple python based data visualization libraries:

1. **Matplotlib**
2. **Seaborn**
3. **Plotly**

etc

1. Basic Introduction to Matplotlib:
----
1. most popular and oldest data visulization library.
Python's alternative to MatLab

2. It is open source and freeware where as Matlab is not open source(closed source)
and not freeware.

3. By using this library we can plot data in graphical form very easily. That graphical form can be either 2-D or 3-D.

4. It is comprehensive library for creating static, animated, and interactive visualizations in python.

5. Jhon Hunter developed matplotlib on top of Numpy and Sidepy libraries.

6. It has very large community support. Every data scientist used this library atleast once in his life.

7. advanced libraries like seaborn, plotly are developed on top of matplotlib.

The official website: https://matplotlib.org

******Examples tab**

Installing Matplotlib:

There are 2 ways

1. with Anaconda distribution, this library will be available automatically.
```
conda install matplotlib
```

2. In our system, if python is already available, then we can install by using python package manager(pip)
```
pip install matplotlib
```

How to check installation:

```
>>> import matplotlib
>>> matplotlib.__version__
'3.4.2'
```

```
D:\durgaclasses>pip list
D:\durgaclasses>pip freeze
```

Types of Plots:

There are multiple types are available to represent our data in graphical form. The important are:

1. Line Plots
2. Bar charts
3. Pie charts
4. Histogram
5. Scatter plots
etc

Based on input data and requirement, we can choose the corresponding plot.

Note:
1. Matplotlib --->package/library
2. pyplot --->module name
3. pyplot module defines several functions to create plots
```
   plot()
   bar()
   pie()
   hist()
   scatter()
   etc
```
4. We can create plots in 2 approaches

1. **Functional oriented approach (For small data sets)**
2. **Object oriented approach (For larger data sets)**

11-07-2021

Line Plots:

We can mark data points from the input data and we can connect these data points with lines. Such type of plots are called line plots.

We can use line plots to determine the relationship between two data sets.
Data set is a collection of values like ndarray,python's list etc

wickets = [1,2,3,4,5,6,7,8,9,10]
overs = [1,4,5,,,..20]

The values from each data set will be plotted along an axis.(x-axis,y-axis)

matplotlib.pyplot.plot()

import matplotlib.pyplot as plt
plt.plot()--->To create line plot
plt.bar()--->To create bar chart
plt.pie()--->To create pie chart

matplotlib--->seaborn and plotly
line plot--->bar graph,pie chart, histogram etc

plot(*args, scalex=True, scaley=True, data=None, **kwargs)

```
plot(*args, scalex=True, scaley=True, data=None, **kwargs)
    Plot y versus x as lines and/or markers.

    Call signatures::

        plot([x], y, [fmt], *, data=None, **kwargs)
        plot([x], y, [fmt], [x2], y2, [fmt2], ..., **kwargs)

    >>> plot(x, y)        # plot x and y using default line style and color
    >>> plot(x, y, 'bo')  # plot x and y using blue circle markers
    >>> plot(y)           # plot y using x as index array 0..N-1
    >>> plot(y, 'r+')     # ditto, but with red plusses
```

***args--->any collection of values(this collection of values will become tuple)**
****kwargs--->any collection of key-value pairs (dictionary)**

plt.plot(x,y)
The data points will be considered from x and y values.
x=[10,20,30]
y=[1,2,3]

data points: (10,1), (20,2),(30,3)

eg-1: Creation of line plot by passing 2 nd-arrays:

```
-----------------------------------------------------
import matplotlib.pyplot as plt
import numpy as np
x = np.arange(1,11)
```

```
y = x**2
plt.plot(x,y) #(1,1),(2,4),(3,9)...
plt.show()
```

What is figure?

Figure is an individual window on the screen, in which matplotlib displays the graphs. ie it is the container for the graphical output.

plot() function is responsible to create this figure object.

How to add title to the line plot:

The title describes the information about our graph.

```
plt.title('Square function line plot')
```

How to add xlabel and ylabel to the line plot:

The xlabel describes inforamtion about x-axis data.
similarly ylabel also

```
plt.xlabel('N value')
plt.ylabel('Square of N')
```

```
plt.xlabel('Overs')
plt.ylabel('Wickets')
plt.title('Fall of wickets')
plt.xlabel('Year')
plt.ylabel('Sales')
plt.title('Nokia Mobile Sales Report')
```

```
import matplotlib.pyplot as plt
import numpy as np
x = np.arange(1,11)
y = x**2
plt.plot(x,y) #(1,1),(2,4),(3,9)...
plt.title('Square Function Line Plot')
plt.xlabel('N value')
plt.ylabel('Square of N')
plt.show()
```

```
plt.plot(x,y)
plt.title()
plt.xlabel()
plt.ylabel()
plt.show()
```

Line properties:

A line drawn on the graph has several properties like color,style,width of the line,transparency etc. We can customize these based on our requirement.

1. Marker property:

We can use marker property to highlight data points on the line plot.
We have to use marker keyword argument.

plt.plot(a,b,marker='o')

o means circle marker

The allowed markers are:
****Markers****

character	description
``'.'``	point marker
``','``	pixel marker
``'o'``	circle marker
``'v'``	triangle_down marker
``'^'``	triangle_up marker
``'<'``	triangle_left marker
``'>'``	triangle_right marker
``'1'``	tri_down marker
``'2'``	tri_up marker
``'3'``	tri_left marker
``'4'``	tri_right marker
``'8'``	octagon marker
``'s'``	square marker
``'p'``	pentagon marker
``'P'``	plus (filled) marker
``'*'``	star marker
``'h'``	hexagon1 marker
``'H'``	hexagon2 marker
``'+'``	plus marker
``'x'``	x marker

```
   ``'X'``        x (filled) marker
   ``'D'``        diamond marker
   ``'d'``        thin_diamond marker
   ``'|'``        vline marker
   ``'_'``        hline marker
   =============  ===============================
```

```
import matplotlib.pyplot as plt
import numpy as np
x = np.arange(1,11)
y = x**2
plt.plot(x,y,marker='*') #(1,1),(2,4),(3,9)...
plt.title('Square Function Line Plot')
plt.xlabel('N value')
plt.ylabel('Square of N')
plt.show()
```

2. Linestyle property:

Linestyle specifies whether the line is solid or dashed or dotted etc
We can specify linestyle by using linestyle keyword argument.

```
-    ----->solid line
--   ----->dashed line
:    ----->dotted line
-.   ----->dash-dotted line
```

```
   =============  ===============================
   character      description
   =============  ===============================
   ``'-'``        solid line style
```

``'--'``	dashed line style
``'-.'``	dash-dot line style
``':'``	dotted line style

```
import matplotlib.pyplot as plt
import numpy as np
x = np.arange(1,11)
y = x**2
plt.plot(x,y,marker='o',linestyle='-.') #(1,1),(2,4),(3,9)...
plt.title('Square Function Line Plot')
plt.xlabel('N value')
plt.ylabel('Square of N')
plt.show()
```

3. color property:

We can specify our required color for the line plot.
We have to use color keyword argument.
We can use any color even hexa code also.
Matplotlib defines some short codes for commonly used colors. We can use short codes also.

character	color
``'b'``	blue
``'g'``	green
``'r'``	red
``'c'``	cyan
``'m'``	magenta
``'y'``	yellow
``'k'``	black

```
    ``'w'``         white
    =============    =================================
```

```
import matplotlib.pyplot as plt
import numpy as np
x = np.arange(1,11)
y = x**2
plt.plot(x,y,marker='o',linestyle='-',color='#b02f15') #(1,1),(2,4),(3,9)...
plt.title('Square Function Line Plot')
plt.xlabel('N value')
plt.ylabel('Square of N')
plt.show()
```

default color:

If we are not specifying color then default color will be selcted from style cycle.
We can check default colors as follows:

```
>>> plt.rcParams['axes.prop_cycle'].by_key()
{'color': ['#1f77b4', '#ff7f0e', '#2ca02c', '#d62728', '#9467bd', '#8c564b',
'#e377c2', '#7f7f7f', '#bcbd22', '#17becf']}
```

first default color: blue
second default color: orange
third default color: green
fourth default color: red
etc

shortcut way to specify all 3 marker, linestyle,color properties:
--

We can use short notation: either mlc or cml

```
plt.plot(a,b,marker='o',linestyle='-',color='r')
plt.plot(a,b,'o-r') #mlc
plt.plot(a,b,'ro-') #cml
```

In this shortcut, we should use shortcode only for color.

```
plt.plot(a,b,'o-r')--->valid
plt.plot(a,b,'o-red')--->invalid
plt.plot(a,b,'o-#1c203d')--->invalid
```

We can see all line properties from the following link:
https://matplotlib.org/2.0.2/api/lines_api.html

```
import matplotlib.pyplot as plt
import numpy as np
a = np.arange(1,11)
plt.plot(a,a**3,'o-r',lw=10,ms=15,mfc='yellow',alpha=0.1) #mlc
plt.show()
```

Components of Line plot:
----
figure
axes/plot
x-axis and y-axis

title
xlabel and ylabel
xticks and yticks

Sequence of activities of plot function:
----
1. Creation of figure object
2. Creation of plot object/axes
3. Draw x and y axis
4. Mark evenly spaced values on x-axis and y-axis(xticks and yticks)
5. Plot the data points
6. Connect these data points with line
7. Add title,xlabel and ylabel

How to customize the size of the figure:

The default size of the figure: 8 inches width and 6 inches height.
But we can customize based on our requirement. For this we have to use figure() function.

```
figure(num=None, figsize=None, dpi=None, facecolor=None, edgecolor=None,
frameon=True, FigureClass=<class 'matplotlib.figure.Figure'>, clear=False,
**kwargs)
   Create a new figure, or activate an existing figure.


figsize : (float, float), default: :rc:`figure.figsize`
      Width, height in inches.
```

```
import matplotlib.pyplot as plt
import numpy as np
#plt.figure(num=1,figsize=(10,4),facecolor='blue')
#plt.figure(num=1,figsize=(3,3),facecolor='g')
plt.figure(figsize=(3,3),facecolor='g')
a = np.arange(1,6)
plt.plot(a,a,'o-r')
plt.show()
```

How to save line plot to a file:

We can save line plot to a file instead of displaying on the screen.
We have to use savefig() function.

```
import matplotlib.pyplot as plt
import numpy as np
#plt.figure(num=1,figsize=(10,4),facecolor='blue')
#plt.figure(num=1,figsize=(3,3),facecolor='g')
plt.figure(figsize=(3,3),facecolor='g')
a = np.arange(1,6)
plt.plot(a,a,'o-r')
plt.savefig('identitylineplot.png')
```

Bydefault this figure will be saved in the current working directory. But we can provide any location based on our requirement.

```
import matplotlib.pyplot as plt
import numpy as np
#plt.figure(num=1,figsize=(10,4),facecolor='blue')
#plt.figure(num=1,figsize=(3,3),facecolor='g')
plt.figure(figsize=(3,3),facecolor='g')
a = np.arange(1,6)
plt.plot(a,a,'o-r')
plt.savefig('C:\\Users\\lenovo\\Desktop\\identitylineplot.jpeg')
```

Creation of line plot by passing a single ndarray:
---
plt.plot(a,b) ----> a for x-axis and b for y-axis.
plt.plot(a) ---->a is for y-axis and x-axis values will be generated automatically by matplotlib from 0 to N-1

```
a = np.array([10,20,30,40,50])
plt.plot(a) # 0 to 4 will be considered for x-axis.
```
Now the data points are: (0,10),(1,20),(2,30),(3,40),(4,50)

```
import matplotlib.pyplot as plt
import numpy as np
a = np.array([10,20,30,40,50])
plt.plot(a,'o-r')
plt.show()
```

Multiple lines on the same plot:

We can plot any number of line plots on the same graph.
This approach is helpful for comparison purpose.

```
import matplotlib.pyplot as plt
import numpy as np
x = np.arange(1,11)
i = x
s = x**2
c = x**3
plt.plot(x,i,'o-r')
plt.plot(x,s,'o-b')
plt.plot(x,c,'o-g')
plt.title('One Graph, but multiple plots')
plt.show()
```

shortcut way:

We can also use single plot() function for all 3 lines.

```
import matplotlib.pyplot as plt
import numpy as np
x = np.arange(1,11)
i = x
s = x**2
c = x**3
plt.plot(x,i,'o-r',x,s,'o-b',x,c,'o-g')
plt.title('One Graph, but multiple plots')
plt.show()
```

Note:

```
plt.plot(x,i,'o-r',x,s,'o-b',x,c,'o-g',lw=10)
```

For the first line: x,i,'o-r'

For the second line: x,s,'o-b'

For the third line: x,c,'o-g'

linewidth property is applicable for all 3 lines.

```
plt.title('Square function line plot')
plt.xlabel('x value')
plt.ylabel('square of x')
```

How to customize title properties:

```
title(label, fontdict=None, loc=None, pad=None, *, y=None, **kwargs)
   Set a title for the Axes.

label : str
      Text to use for the title

fontdict : dict
      A dictionary controlling the appearance of the title text

loc : {'center', 'left', 'right'}
pad : float, default: :rc:`axes.titlepad`
      The offset of the title from the top of the Axes, in points.

**kwargs --->To customize font properties
```

TEXT properties of matplotlib:

https://matplotlib.org/stable/tutorials/text/text_props.html

```
family          [ 'serif' | 'sans-serif' | 'cursive' | 'fantasy' | 'monospace' ]
style or fontstyle   [ 'normal' | 'italic' | 'oblique' ]
weight or fontweight      [ 'normal' | 'bold' | 'heavy' | 'light' | 'ultrabold' |
'ultralight']
```

```
import matplotlib.pyplot as plt
import numpy as np
x = np.arange(1,11)
s = x**2
plt.plot(x,s,'o-r')
plt.title('Square Function
Plot',{'color':'b','size':20,'backgroundcolor':'yellow','alpha':1,'fontstyle':'italic','fa
mily':'cursive','weight':1000,'rotation':1},loc='left',pad=25,color='red')
```

plt.show()

https://matplotlib.org/stable/tutorials/text/text_props.html

Customization of xlabel and ylabel:
----
exactly same as title customization only

```
xlabel(xlabel, fontdict=None, labelpad=None, *, loc=None, **kwargs)
   Set the label for the x-axis.

ylabel(ylabel, fontdict=None, labelpad=None, *, loc=None, **kwargs)
   Set the label for the y-axis.
```

```
import matplotlib.pyplot as plt
import numpy as np
a = np.arange(1,11)
b = a**2
plt.plot(a,b)
plt.title('Square Function Line plot')
plt.xlabel('Year',{'color':'r','size':20,'backgroundcolor':'yellow','rotation':10,'alph
a':1,'fontstyle':'italic','family':'cursive','weight':1000})
plt.ylabel('Data Science
Sales',color='r',size=20,backgroundcolor='yellow',rotation=100,alpha=1,fontstyl
e='italic',family='cursive',weight=1000)
plt.show()
```

Note: {'color':'r'} and

color='b'
In the case of conflict keyword arguments will get more priority.

Note: fontdict properties are same for title,xlabel and ylabel. These values can be passed as keyword arguments also. In the case of conflict, keyword arguments will get more priority.

How to add grid lines to the plot:

We can add grid lines to the plot. For this we have to use grid() function.
plt.grid()

```
grid(b=None, which='major', axis='both', **kwargs)
   Configure the grid lines.


b : bool or None, optional
      Whether to show the grid lines. If any *kwargs* are supplied,
      it is assumed you want the grid on and *b* will be set to True.

      If *b* is *None* and there are no *kwargs*, this toggles the
      visibility of the lines.
```

```
plt.grid()--->on-->grid lines are visible
plt.grid()-->off-->grid lines are invisible
plt.grid()--->on-->grid lines are visible
plt.grid()-->off-->grid lines are invisible
```

case-1:
plt.grid()
In this case grid will be visible.

case-2:
plt.grid()
plt.grid()

grid lines won't be visible

case-3:
plt.grid()
plt.grid(color='g')

grid lines are visible

case-4:
plt.grid(b=True)
plt.grid(b=False)

grid lines are invisible

which property:
----
major grid lines and minor grid lines
It decides which grid lines have to display whether major or minor
The allowed values:

which : {'major', 'minor', 'both'}
The default value is major.

>>> help(plt.minorticks_on)
Help on function minorticks_on in module matplotlib.pyplot:

minorticks_on()

Display minor ticks on the axes.

Displaying minor ticks may reduce performance; you may turn them off using `minorticks_off()` if drawing speed is a problem.

```
import matplotlib.pyplot as plt
import numpy as np
a = np.array([10,20,30,40,50])
plt.plot(a,a,'o-r')
plt.minorticks_on()
plt.grid(which='both')
plt.show()
```

Difference between major and minor grid lines:

```
import matplotlib.pyplot as plt
import numpy as np
a = np.array([10,20,30,40,50])
plt.plot(a,a,'o-r',lw=7,markersize=10,mfc='yellow')
plt.grid(color='red',lw=3)
plt.minorticks_on()
plt.grid(which='minor',color='g')
plt.show()
```

axis property:

Along which axis, grid lines have to display

axis : {'both', 'x', 'y'},
default value: both

```
import matplotlib.pyplot as plt
import numpy as np
a = np.array([10,20,30,40,50])
plt.plot(a,a,'o-r',lw=7,markersize=10,mfc='yellow')
plt.grid(axis='y')
plt.show()
```

Note: We can use several keyword arguments also.
plt.grid(color='g',lw=2,linestyle=':')

```
import matplotlib.pyplot as plt
import numpy as np
a = np.array([10,20,30,40,50])
plt.plot(a,a,'o-r',lw=7,markersize=10,mfc='yellow')
plt.grid(color='g',lw=2,linestyle=':')
plt.show()
```

Adding Legend:

If multiple lines present then it is difficult to identify which line represents which dataset/function.

To overcome this problem we can add legend.
plt.legend()

Syntax:

```
legend(*args, **kwargs)

Call signatures::

        legend()
        legend(labels)
        legend(handles, labels)
```

1. legend():
----
entries will be added to the legend in the order of plots creation.

```
import matplotlib.pyplot as plt
import numpy as np
a = np.arange(10)
plt.plot(a,a,marker='o',label='identity')
plt.plot(a,a**2,marker='o',label='square')
plt.plot(a,a**3,marker='o',label='cubic')
plt.legend()
plt.show()
```

2. legend(labels)

The argument is list of strings.
Each string is considered as a lable for the plots, in the order they created.

```
plt.legend(['label-1','label-2','label-3'])
```

This approach is best suitable for adding legend for already existing plots.

```
import matplotlib.pyplot as plt
import numpy as np
a = np.arange(10)
plt.plot(a,a,marker='o')
plt.plot(a,a**2,marker='o')
plt.plot(a,a**3,marker='o')
plt.legend(['identity','square','cubic'])
plt.show()
```

Note: This approach is not recommended to use because we should aware the order in which plots were created.

legend(handles, labels):

We can define explicitly lines and labels in the legend() function itself.
It is recommended approach as we have complete control.

```
plt.legend([line1,line2,line3],['label-1','label-2','label-3'])
```

observation:

```
l = [10]
a = l
print(a) #[10]

a, = l #unpack list elements and then assign values ot provided variables
print(a) #10
```

```
plt.plot(x,i,'o-r',x,s,'o-b',x,c,'o-g',lw=10)
```

For the first line: x,i,'o-r'
For the second line: x,s,'o-b'
For the third line: x,c,'o-g'

```
import matplotlib.pyplot as plt
import numpy as np
a = np.arange(10)
line1, = plt.plot(a,a,'o-r')
line2, = plt.plot(a,a**2,'o-b')
line3, = plt.plot(a,a**3,'o-g')
plt.legend([line1,line3,line2],['identity','cubic','square'])
plt.show()
```

How to adjust legend location:

Based on our requirement we can decide legend location in the plot.
loc argument
loc--->location

The possible values for the location are:

```
 ===============   =============
Location String   Location Code
===============   =============
'best'           0
'upper right'     1
'upper left'      2
'lower left'      3
'lower right'     4
'right'           5
'center left'     6
```

'center right'	7
'lower center'	8
'upper center'	9
'center'	10

```
import matplotlib.pyplot as plt
import numpy as np
a = np.arange(10)
lines = plt.plot(a,a,'o-r',a,a**2,'o-b',a,a**3,'o-g')
plt.legend(lines,['identity','square','cubic'],loc = 10)
plt.show()
```

How to specify number of columns in the legend:

Bydefault the number of columns: 1
But we can customize by using ncol argument.

```
import matplotlib.pyplot as plt
import numpy as np
a = np.arange(10)
lines = plt.plot(a,a,'o-r',a,a**2,'o-b',a,a**3,'o-g')
plt.legend(lines,['identity','square','cubic'],ncol=3)
plt.show()
```

We can do more customization for the legend like

1. We can add title to the legend.
2. We can change look and feel
3. We can change fontsize and color
4. We can place legend outside of the plot
etc

Adding title to the legend:

We can title for the legend explicitly. For this we have to use title keyword argument.

```
import matplotlib.pyplot as plt
import numpy as np
a = np.arange(10)
lines = plt.plot(a,a,'o-r',a,a**2,'o-b',a,a**3,'o-g')
plt.legend(lines,['identity','square','cubic'],title='3 common functions')
plt.show()
```

Diagram: legend_title

How to add legend outside of the plot:

We can add legend outside of the plot also.
For this we have to use loc keyword argument.
loc = (x,y)

Diagram: legend_loc1

```
import matplotlib.pyplot as plt
import numpy as np
a = np.arange(10)
lines = plt.plot(a,a,'o-r',a,a**2,'o-b',a,a**3,'o-g')
plt.legend(lines,['identity','square','cubic'],loc=(0,1.1))
plt.tight_layout()
plt.show()
```

Diagram: legend_loc2

Customization of tick location and labels:

--

Ticks are the markers to represent specific value on the axis.
Ticks are very helpful to locate data points on the plot very easily.
Based on our input, matplotlib decides tick values automatically.
Based on our requirement, we can customize tick location and labels.
For this we have to use

```
   xticks()
   yticks()
```

```
xticks(ticks=None, labels=None, **kwargs)
   Get or set the current tick locations and labels of the x-axis.

   Pass no arguments to return the current values without modifying them.
```

getter method--->to get data
setter method-->to set our own data

```
plt.xticks(ticks=[0,1,2,3,4,5,6,7,8,9,10]) #to place our own xtick values
```

For these xticks we can add labels also

```
plt.xticks([0,1,2,3,4,5,6,7,8,9,10],['2000','2001','2002','2003','2004','2005','2006',
'2007','2008','2009','2010']) #to place our own xtick values
```

We can customize label properties by using keyword arguments like color,font,size etc

```
import matplotlib.pyplot as plt
import numpy as np
```

```
a = np.arange(11)
b = a*100
plt.plot(a,b,'o-r')
plt.grid()
plt.title('Sales Report')
plt.xlabel('Year')
plt.ylabel('Number of sales')
plt.xticks([0,1,2,3,4,5,6,7,8,9,10],['2000','2001','2002','2003','2004','2005','2006',
'2007','2008','2009','2010'],color='blue',size=15,rotation=30) #to place our own
xtick values
plt.show()
```

Similarly we can customize yticks also.

```
plt.yticks([0,100,200,300,400,500,600,700,800,900,1000])
```

```
import matplotlib.pyplot as plt
import numpy as np
a = np.arange(11)
b = a*100
plt.plot(a,b,'o-r')
plt.grid()
plt.title('Sales Report')
plt.xlabel('Year')
plt.ylabel('Number of sales')
plt.xticks([0,1,2,3,4,5,6,7,8,9,10],['2000','2001','2002','2003','2004','2005','2006',
'2007','2008','2009','2010'],color='blue',size=15,rotation=30) #to place our own
xtick values
plt.yticks([0,500,1000])
plt.show()
```

Note:

1. Without providing tick values we cannot provide labels, otherwise we will get error.

```
import matplotlib.pyplot as plt
import numpy as np
a = np.arange(11)
b = a*100
plt.plot(a,b,'o-r')
plt.grid()
plt.title('Sales Report')
plt.xlabel('Year')
plt.ylabel('Number of sales')
plt.xticks(labels=['2000','2001','2002','2003','2004','2005','2006','2007','2008','20
09','2010'],color='blue',size=15,rotation=30) #to place our own xtick values
plt.yticks([0,500,1000])
plt.show()
```

```
TypeError: xticks(): Parameter 'labels' can't be set without setting 'ticks'
```

2. If we pass empty list to ticks then tick values will become invisible.

```
plt.yticks([])
```

How to set limit range of values on x-axis and y-axis:

---

xlim() and ylim() functions

for x-axis:

left

right

For y-axis:

bottom

top

```
xlim(*args, **kwargs)
   Get or set the x limits of the current axes.

   Call signatures::

      left, right = xlim()  # return the current xlim
      xlim((left, right))   # set the xlim to left, right
      xlim(left, right)     # set the xlim to left, right

   If you do not specify args, you can pass *left* or *right* as kwargs,
   i.e.::

      xlim(right=3)  # adjust the right leaving left unchanged
      xlim(left=1)  # adjust the left leaving right unchanged
```

If we are not passing any argument to xlim() function then it acts as getter function.
If we are passing any argument then it acts as setter function.

Getting left and right limits on the x-axis:

```
import matplotlib.pyplot as plt
import numpy as np
a = np.arange(1,101)
b = a**2
plt.plot(a,b,'o-r')
plt.grid()
left,right = plt.xlim()
bottom,top = plt.ylim()
print('Left limit on the x-axis:',left)
print('Right limit on the x-axis:',right)
print('Bottom limit on the y-axis:',bottom)
print('Top limit on the y-axis:',top)
plt.show()
```

```
Left limit on the x-axis: -3.95
Right limit on the x-axis: 104.95
Bottom limit on the y-axis: -498.95000000000005
Top limit on the y-axis: 10499.95
```

To set left and right limits on x-axis:

```
plt.xlim(left,right)
plt.xlim((left,right))
plt.xlim(right=3) left will be generated by matplotlib
plt.xlim(left=3) right will be generated by matplotlib
plt.xlim(3)----->3 is for left and default for right
```

```
import matplotlib.pyplot as plt
import numpy as np
a = np.arange(1,101)
b = a**2
plt.plot(a,b,'o-r')
plt.grid()
#plt.xlim(1,50) # left is 1 and right is 50
#plt.xlim(left=1) #left is 1 and for right default value
#plt.xlim(right=50) #left is default and for right 50
plt.xlim(1)
plt.show()
```

ylim() function:

```
import matplotlib.pyplot as plt
import numpy as np
a = np.arange(1,101)
b = a**2
plt.plot(a,b,'o-r')
plt.grid()
plt.ylim(bottom=100)
print(plt.ylim())
plt.show()
```

How to set limits for x-axis and y-axis
How to customize xticks and yticks

on the x-axis: 0 to 50   0-->left,50-->right
on the y-axis: 10 to 100 , 10--->bottom and 100---->top

Scaling: How to set scale for x-axis and y-axis:
--
The difference between any two consecutive points on any axis is called scaling.

The most commonly used scales are:

1. Linear scaling
2. Logarithmic Scaling

1. Linear scaling:

The difference between any two consecutive points on the given axis is always fixed, such type of scaling is called linear scaling.
Default scaling in matplotlib is linear scaling.

If the data set values are spreaded over small range, then linear scaling is the best choice.

2. Logarithmic Scaling:

The difference between any two consecutive points on the given axis is not fixed and it is multiples of 10, such type of scaling is called logarithmic scaling.

If the data set values are spreaded over big range, then logarithmic scaling is the best choice.

```
plt.xticks()
plt.yticks()
plt.xlim()
plt.ylim()


plt.xscale()
plt.yscale()

xscale(value, **kwargs)
   Set the x-axis scale.
value : {"linear", "log", "symlog", "logit", ...}

yscale(value, **kwargs)
   Set the y-axis scale.
value : {"linear", "log", "symlog", "logit", ...}

import matplotlib.pyplot as plt
import numpy as np
a = np.arange(10000)
b = np.arange(10000)
plt.plot(a,b)
plt.grid()
#plt.xscale('linear')
plt.xscale('log')
plt.show()
```

Linear scaling for x-axis and logarithmic scaling for y-axis:

--

```
import matplotlib.pyplot as plt
import numpy as np
a = np.arange(10000)
b = np.arange(10000)
plt.plot(a,b)
plt.grid()
plt.xscale('linear') #linear scaling
plt.yscale('log') #logarithmic scaling
plt.show()
```

For both x-axis and y-axis, logarithmic scaling:

```
import matplotlib.pyplot as plt
import numpy as np
a = np.arange(10000)
b = np.arange(10000)
plt.plot(a,b)
plt.grid()
plt.xscale('log') #logarithmic scaling
plt.yscale('log') #logarithmic scaling
plt.show()
```

How to customize base value in logarithmic scaling:

We have to use keyword argument base

```
import matplotlib.pyplot as plt
import numpy as np
a = np.arange(10000)
b = np.arange(10000)
plt.plot(a,b)
plt.grid()
plt.xscale('log',base=2) #logarithmic scaling
plt.yscale('log',base=9) #logarithmic scaling
plt.show()
```

plotting styles:

We can customize look and feel of hte plot by using style library.
There are multiple predefined styles are available...

```
plt.style.available
```

```
>>> plt.style.available
['Solarize_Light2', '_classic_test_patch', 'bmh', 'classic', 'dark_background',
'fast', 'fivethirtyeight', 'ggplot', 'grayscale', 'seaborn', 'seaborn-bright', 'seaborn-
colorblind', 'seaborn-dark', 'seaborn-dark-palette', 'seaborn-darkgrid', 'seaborn-
deep', 'seaborn-muted', 'seaborn-notebook', 'seaborn-paper', 'seaborn-pastel',
'seaborn-poster', 'seaborn-talk', 'seaborn-ticks', 'seaborn-white', 'seaborn-
whitegrid', 'tableau-colorblind10']
```

matplotlib--->one visuliazation library
seaborn -->another visuliazation library

Note:
1. ggplot--->To emulate the most powerful ggplot style of R language.

2. seaborn-->To emulate seaborn style
3. fivethirtyeight--->The most commonly used style in real time.
etc

We can set our own customized style for the plot as follows:
plt.style.use('ggplot')

```
import matplotlib.pyplot as plt
import numpy as np
a = np.arange(10000)
b = np.arange(10000)
#plt.style.use('ggplot')
#plt.style.use('seaborn')
plt.style.use('grayscale')
plt.plot(a,b)
plt.show()
```

Procedural/Functional oriented vs Object Oriented Approaches of plotting:
--
2 approaches to create plot

1. Procedural/Functional oriented
2. OOP

procedural:

```
def f1():
	print('f1 function')

def f2():
	print('f2 function')

def f3():
        print('f31 function')

def f4():
        print('f4 function')

f1()
f2()
f3()
f4()
```

OOP approach:

```
class Test:
        def m1(self):
                print('m1 method')
        def m2(self):
                print('m2 method')
        def m3(self):
                print('m3 method')
        def m4(self):
                print('m4 method')

t = Test()
t.m1()
t.m2()
t.m3()
t.m4()
```

1. Procedural/Functional Approach:

We can create plots with the help of mulitple functions from pyplot module.

```
#Creation of line plot to represent square functionality from 1 to 10.
import matplotlib.pyplot as plt
import numpy as np
a = np.arange(1,11)
b = a**2
plt.plot(a,b)
plt.xlabel('N')
plt.ylabel('Square Value of N')
plt.title('Square Function')
plt.show()
```

plot()
xlabel()
ylable()
title()
show()

2. Object Oriented Approach:

In this approach, we have to create objects and on those objects we have to call corresponding methods to create a plot.

1. Creation of Figure object:
----
fig = plt.figure()

```
figure(num=None, figsize=None, dpi=None, facecolor=None, edgecolor=None,
frameon=True, FigureClass=<class 'matplotlib.figure.Figure'>, clear=False,
**kwargs)
   Create a new figure, or activate an existing figure.
```

fig = plt.figure()

2. Creation of Axes object:

Once figure object is ready, then we have to add axes to that object. For this we have to use add_axes() method of Figure class. This method returns Axes object.

```
add_axes(self, *args, **kwargs)
   Add an Axes to the figure.

   Call signatures::

      add_axes(rect, projection=None, polar=False, **kwargs)
      add_axes(ax)

rect : sequence of float
      The dimensions [left, bottom, width, height] of the new Axes. All
      quantities are in fractions of figure width and height.
```

3. plotting the graph and setting axes properties:

--

Once Axes object is ready, then we can use the following methods.

```
axes.plot(a,b)
axes.set_xlabel('xlabel')
axes.set_ylabel('ylabel')
axes.set_title('title')
plt.show()
```

```
#Creation of line plot to represent square functionality from 1 to 10.
import matplotlib.pyplot as plt
import numpy as np
a = np.arange(1,11)
b = a**2
fig = plt.figure()
axes = fig.add_axes([0.2,0.3,0.6,0.4]) #[left,bottom,width,height] lbwh
axes.plot(a,b)
axes.set_xlabel('N')
axes.set_ylabel('Square of N')
axes.set_title('Square Function')
axes.grid()
plt.show()
```

Note: We can use single set() method to set all axes properties like title,xlabel,ylabel,xlim,ylim etc

```
axes.set(xlabel='N',ylabel='Square of N', title='Square
Function',xlim=(1,5),ylim=(1,25))
```

```
#Creation of line plot to represent square functionality from 1 to 10.
import matplotlib.pyplot as plt
import numpy as np
a = np.arange(1,11)
b = a**2
fig = plt.figure()
axes = fig.add_axes([0.2,0.3,0.6,0.4]) #[left,bottom,width,height] lbwh
axes.plot(a,b)
axes.set(xlabel='N',ylabel='Square of N', title='Square
Function',xlim=(1,5),ylim=(1,25))
axes.grid()
plt.show()
```

Summary:
----
1. Creation of Figure object
2. Creation of Axes object
3. plot the graph
4. set the properties of the axis.

Bar Chart/Bar Graph/Bar Plot:

In a line plot, the data points will be marked and these markers will be connected by line.

But in bar chart, data will be represented in the form of bars.

4 types of bar charts

1. Simple bar chart/vertical bar chart
2. Horizontal bar chart
3. Stacked Bar chart
4. Clustered Bar Chart/Grouped Bar Chart

1. Simple bar chart/vertical bar chart:

The data will be represented in the form of vertical bars.
Each vertical bar represents an individual category.
The height/length of the bar is based on value it represents.
Most of the times the width of the bar is fixed, but we can customize.

The default width: 0.8

By using bar() function we can create.

plt.bar()

Syntax:

```
-------
bar(x, height, width=0.8, bottom=None, *, align='center', data=None, **kwargs)
Make a bar plot.

x--->values of x-axis, category names
height--->values for y-axis, height of the bars.
width-->width of each bar,default is 0.8
bottom-->From where bar has to start.
align-->alignment of the bars on the x-axis.
   align : {'center', 'edge'}, default: 'center'
    Alignment of the bars to the *x* coordinates:

    - 'center': Center the base on the *x* positions.
    - 'edge': Align the left edges of the bars with the *x* positions.

    To align the bars on the right edge pass a negative *width* and
    ``align='edge'``.
```

eg-1: Represent the number of movies of each hero by using bar chart

```
--------------------------------------------------------------------
import matplotlib.pyplot as plt

heroes = ['Prabhas','Pawan','Chiranjeevi','Sharukh','Amitabh'] # x-axis values
movies = [100,300,200,600,1000] #height of bars, values for y-axis
```

```
plt.bar(heroes,movies)

plt.xlabel('Hero Name',color='b',fontsize=15)
plt.ylabel('Number of Movies',color='b',fontsize=15)
plt.title('Hero wise number of movies',color='r',fontsize=15)
plt.show()
```

We can customize several things like

1. changing color of each bar
2. changing width of each bar
3. changing bottom of each bar
4. changing alignments
etc

Observations:
----
1. plt.bar(heroes,movies,color='r')
  Now all bars with RED color

2. Separate color for each bar

```
c = ['r','b','k','g','orange']
plt.bar(heroes,movies,color=c)
```

3. The width of each bar should be 0.5(default is 0.8)

```
plt.bar(heroes,movies,width=0.5       )
```

4. Different widths for bars

```
w = [0.8,0.6,0.7,0.9,0.5]
plt.bar(heroes,movies,width=w)
```

5. bottom should be 50 instead of 0

```
plt.bar(heroes,movies,bottom=50)
```

6. Different bottom values for bar?

```
b=[0,10,30,50,70]
plt.bar(heroes,movies,bottom=b)
```

7. alignment: center

for left alignment:

```
  plt.bar(heroes,movies,align='edge')
```

for right alignment:

```
 plt.bar(heroes,movies,width=-0.8,align='edge')
```

eg-2: Mobile Sales of Nokia Company from 2011 to 2020:

```
import matplotlib.pyplot as plt

years = [2011, 2012, 2013, 2014, 2015, 2016, 2017, 2018, 2019, 2020]
sales = [10000, 25000, 45000, 30000, 10000, 5000,70000,60000,65000,50000]

c = ['r','k','y','g','orange','m','c','b','lime','violet']
plt.bar(years,sales,color=c)

plt.xlabel('Year',color='b',fontsize=15)
plt.ylabel('Number of Sales',color='b',fontsize=15)
plt.title('Nokia Mobile Sales in the last Decade',color='r',fontsize=15)
plt.xticks(years,rotation=30)
plt.tight_layout()
plt.grid(axis='y')
plt.show()
```

How to add labels to the bar:

We can add labels to any plot by using the following 2 functions

1. pyplot.text()
2. pyplot.annotate()

1. pyplot.text():
```
----------------
Syntax:
text(x, y, s, fontdict=None, **kwargs)
   Add text to the Axes.

   Add the text *s* to the Axes at location *x*, *y* in data coordinates.
```

Adding Labels for the data points of lineplot:
```
--------------------------------------------
import matplotlib.pyplot as plt
import numpy as np

a = np.arange(10)
plt.plot(a,a,'o-r')
for i in range(a.size): # 0 to 9
        plt.text(a[i]+0.4,a[i]-0.2,f'({a[i]},{a[i]})',color='b')
plt.show()
```

2. pyplot.annotate():
```
--------------------
annotate(text, xy, *args, **kwargs)
   Annotate the point *xy* with text *text*.
   xy : (float, float)
      The point *(x, y)* to annotate.
```

```
import matplotlib.pyplot as plt
import numpy as np

a = np.arange(10)
plt.plot(a,a,'o-r')
for i in range(a.size): # 0 to 9
        #plt.text(a[i]+0.4,a[i]-0.2,f'({a[i]},{a[i]})',color='b')
        plt.annotate(f'({a[i]},{a[i]})',(a[i]+0.4,a[i]-0.2),color='g')
plt.show()
```

How to add labels to the bar chart:

```
import matplotlib.pyplot as plt

years = [2011, 2012, 2013, 2014, 2015, 2016, 2017, 2018, 2019, 2020]
sales = [10000, 25000, 45000, 30000, 10000, 5000,70000,60000,65000,50000]

plt.bar(years,sales,color='r')

plt.xlabel('Year',color='b',fontsize=15)
plt.ylabel('Number of Sales',color='b',fontsize=15)
plt.title('Nokia Mobile Sales in the last Decade',color='r',fontsize=15)
plt.xticks(years,rotation=30)
plt.tight_layout()
for i in range(len(years)): # 0 to 9
        plt.text(years[i],sales[i]+500,sales[i],ha='center',color='b')
plt.show()
```

With more readable labels:

```
import matplotlib.pyplot as plt

years = [2011, 2012, 2013, 2014, 2015, 2016, 2017, 2018, 2019, 2020]
sales = [10000, 25000, 45000, 30000, 10000, 5000,70000,60000,65000,50000]

plt.bar(years,sales,color='r')

plt.xlabel('Year',color='b',fontsize=15)
plt.ylabel('Number of Sales',color='b',fontsize=15)
plt.title('Nokia Mobile Sales in the last Decade',color='r',fontsize=15)
plt.xticks(years,rotation=30)
plt.tight_layout()
for i in range(len(years)): # 0 to 9
	plt.text(years[i],sales[i]+500,str(sales[i]//1000)+'k',ha='center',color='b')
	#plt.annotate(str(sales[i]//1000)+'k',(years[i],sales[i]+500),ha='center',col
or='g',backgroundcolor='yellow')
plt.show()



color =
['#f54287','#f542ec','#bc42f5','#427ef5','#42d7f5','#4287f5','#f56f42','#f2f542','#
5df542','#42f5b6']
```

Plotting bar chart with data from csv file:

Assume that data is available in students.csv file, which is present in current working directory.

```
import matplotlib.pyplot as plt
import numpy as np
import csv
```

```
names = np.array([],dtype='str')
marks = np.array([],dtype='int')

f = open('students.csv','r')
r = csv.reader(f) # Returns csvreader object
h = next(r) #to read header and ignore
for row in r:
        names = np.append(names,row[0])
        marks = np.append(marks,int(row[1]))

plt.bar(names,marks,color='r')
plt.show()
```

Note:
If the labels are too long or too many values to represent then we should go for horizontal bar chart instead of vertical bar chart.

bar()--->to create vertical bar chart
barh() -->to create horizontal bar chart

Horizontal bar chart:

Here the data will be represented in the form of horizontal bars.
Each bar represents an individual category.
The categories will be plotted on y-axis and data values will be plotted on x-axis.
width/length of the bar is proportional to the value it represents.
The default height is 0.8, but we can customize this value.

barh() --->To create horizontal bar chart.

Syntax:
barh(y, width, height=0.8, left=None, *, align='center', **kwargs)
Make a horizontal bar plot.

vertical vs horizontal
----
height ----->width
width ---> height
bottom -->left
bar() -->barh()

students.csv:

Name of Student	Marks
Sunny	100
Bunny	200
Chinny	300
Vinny	200
Pinny	400
Zinny	300
Kinny	500
Minny	600
Dinny	400
Ginny	700
Sachin	300
Dravid	900
Kohli	1000
Rahul	800

Ameer	**600**
Sharukh	**500**
Salman	**700**
Ranveer	**600**
Katrtina	**300**
Kareena	**400**

demo program:

```
import matplotlib.pyplot as plt
import numpy as np
import csv

names = np.array([],dtype='str')
marks = np.array([],dtype='int')

f = open('students.csv','r')
r = csv.reader(f) # Returns csvreader object
h = next(r) #to read header and ignore
for row in r:
	names = np.append(names,row[0])
	marks = np.append(marks,int(row[1]))
plt.barh(names,marks,color='r')
plt.xlabel('Marks',fontsize=15,color='b')
plt.ylabel('Name of Student',fontsize=15,color='b')
plt.title('Students Marks Report',fontsize=15,color='r')
plt.tight_layout()
plt.show()
```

Vertical bar chart
Horizontal bar chart

Stacked Bar chart:

If each category contains multiple subcategories then we should go for stacked bar chart. Here each subcategory will be plotted on top of other subcategory.

eg-1: Country wise total population we have to represent.But in that population we have to plot separately men and women.

eg-2: Country wise medals we have to represent.But in that total number of medals, we have to represent gold,silver and bronze medals separately.

verical bar chart--->bar()
horizontal bar chart--->barh()
stacked bar chart--->either bar() or barh()

The stacked bar chart can be either vertical or horizontal

demo program-1: marks wise student data

```
import matplotlib.pyplot as plt

names = ['Sunny','Bunny','Chinny','Vinny','Tinny']
english_marks = [90,80,85,25,50]
maths_marks = [25,23,45,32,50]
```

```
plt.bar(names,english_marks,color='r')
plt.bar(names, maths_marks, bottom=english_marks, color='green')
plt.show()
```

with labels on the bar:

```
----------------------
import matplotlib.pyplot as plt
import numpy as np
names = ['Sunny','Bunny','Chinny','Vinny','Tinny']
english_marks = np.array([90,80,85,25,50])
math_marks = np.array([25,23,45,32,25])
total_marks = english_marks+math_marks

plt.bar(names,english_marks,color='#09695c',label='English')
plt.bar(names,math_marks,bottom=english_marks,color='#9c0c8b',label="Math
s")

for i in range(len(names)):
        plt.text(names[i],(english_marks[i]/2),str(english_marks[i]),ha='center',co
lor='white',weight=1000)
        plt.text(names[i],(english_marks[i]+math_marks[i]/2),str(math_marks[i]),
ha='center',color='white',weight=1000)
        plt.text(names[i],(total_marks[i]+2),
str(total_marks[i]),ha='center',color='#008080',weight=1000)

plt.xlabel("Name of the Student",color='#570b66', fontsize=15,weight=1000)
plt.ylabel("Marks",color='#570b66', fontsize=15,weight=1000)
plt.title("Student -wise Marks",color='#ED0A3F', fontsize=15,weight=1000)
plt.legend()
plt.tight_layout()
plt.show()
```

Diagram: stacked_bar_with_text_labels

eg-2: country wise medals but sub categories

```
import matplotlib.pyplot as plt
import numpy as np
country_name = ['India','China','US','UK']
gold_medals = np.array([60,40,50,20])
silver_medals = np.array([50,30,25,43])
bronze_medals = np.array([55,24,45,6])
plt.bar(country_name,gold_medals,color='#FFD700',label='gold')
plt.bar(country_name,silver_medals,bottom =
gold_medals,color='#C0C0C0',label='silver')
plt.bar(country_name,bronze_medals,bottom = gold_medals+silver_medals
,color='#CD7F32',label='bronze')
plt.xlabel('Country Name',color='b',fontsize=15)
plt.ylabel('Number of Medals',color='b',fontsize=15)
plt.title('Country Wise Medals Report',color='r',fontsize=15)
plt.legend()
plt.show()
```

eg-3: Stacked Horizontal Bar Chart:

```
import matplotlib.pyplot as plt
import numpy as np
country_name = ['India','China','US','UK']
gold_medals = np.array([60,40,50,20])
```

```
silver_medals = np.array([50,30,25,43])
bronze_medals = np.array([55,24,45,6])
plt.barh(country_name,gold_medals,color='#FFD700',label='gold')
plt.barh(country_name,silver_medals,left =
gold_medals,color='#C0C0C0',label='silver')
plt.barh(country_name,bronze_medals,left = gold_medals+silver_medals
,color='#CD7F32',label='bronze')
plt.ylabel('Country Name',color='b',fontsize=15)
plt.xlabel('Number of Medals',color='b',fontsize=15)
plt.title('Country Wise Medals Report',color='r',fontsize=15)
plt.legend()
plt.show()
```

bar()--->barh()
bottom--->left
xlabel and ylabels are interchanged.

Vertical bar chart
Horizontal bar chart
stacked bar chart

4. Clustered Bar chart/Grouped Bar Chart/Multiple Bar Chart:

If each category contains multiple sub categories and if we want to represent all these sub categories side by side then we should go for Clustered Bar Chart.

eg-1: Country wise total population we have to represent.But in that population we have to plot separately men and women side by side.

eg-2: Country wise medals we have to represent.But in that total number of medals, we have to represent gold,silver and bronze medals separately side by side.

We can create clustered bar chart by using either bar() or barh() functions.

Demo program:

```
-------------
import matplotlib.pyplot as plt
import numpy as np
names = ['Sunny','Bunny','Chinny','Vinny','Tinny']
english_marks = np.array([90,80,85,25,50])
math_marks = np.array([25,23,45,32,25])
xpos = np.arange(len(names)) #[0,1,2,3,4]
w = 0.3
plt.bar(xpos,english_marks,color='r',width=w)
plt.bar(xpos+w,math_marks,color='g',width=w)
#plt.xticks(xpos+0.15,names)
plt.xticks(xpos+w/2,names)
plt.legend(['eng','math'])
plt.show()
```

Demo program-2:

```
---------------
import matplotlib.pyplot as plt
import numpy as np
country_name = ['India','China','US','UK']
gold_medals = np.array([60,40,50,20])
silver_medals = np.array([50,30,25,43])
bronze_medals = np.array([55,24,45,6])
xpos = np.arange(len(country_name)) #[0,1,2,3]
w = 0.2
```

```
plt.bar(xpos,gold_medals,color='#FFD700',width=w)
plt.bar(xpos+w,silver_medals,color='#C0C0C0',width=w)
plt.bar(xpos+2*w,bronze_medals,color='#CD7F32',width=w)

plt.xticks(xpos+w,country_name)
plt.ylabel('Country Name',color='b',fontsize=15)
plt.xlabel('Number of Medals',color='b',fontsize=15)
plt.title('Country Wise Medals Report',color='r',fontsize=15)
plt.legend(['gold','silver','bronze'])

for i in range(len(country_name)):
	plt.text(xpos[i],gold_medals[i]+1,gold_medals[i],ha='center',color='r',wei
ght=1000)
	plt.text(xpos[i]+w,silver_medals[i]+1,silver_medals[i],ha='center',color='r'
,weight=1000)
	plt.text(xpos[i]+2*w,bronze_medals[i]+1,bronze_medals[i],ha='center',col
or='r',weight=1000)

plt.show()
```

eg-2A: India and Australia 20-20 overwise scores required to represent by using clustered bar chart?

```
import matplotlib.pyplot as plt
import numpy as np
overs = np.arange(1,21)
xpos=np.arange(overs.size)
ind_score = [10,12,8,9,10,22,13,17,3,23,11,10,9,8,23,30,10,9,8,7]
aus_score = [6,8,9,15,23,8,9,6,10,17,13,2,21,15,19,17,4,12,14,10]
w=0.3
plt.figure(num=1,figsize=(16,4),facecolor='g')
plt.bar(xpos,ind_score,width=w)
plt.bar(xpos+w,aus_score,width=w)
plt.xticks(xpos+(w/2),labels=overs)
plt.xlabel('Overs',color='b')
plt.ylabel('Number of Runs',color='b')
plt.title('Overwise Scores Summary')
plt.legend(['IND','AUS'],ncol=2)
plt.grid(axis='y')
plt.show()
```