Python for Data Science

Machine Learning & Deep Learning

By DURGA SIR

NumPy Course

Q1. What is the need of Numpy?

Q2. What is Numpy and History?

Q3. In Which language Numpy was written?

Q4. What is the purpose of Numpy?

Q5. How to install Numpy?

Q6. What are various Features of Numpy?

Q7. What is nd array in Numpy?

Q8. What is the advantage of numpy array over Python's in built List?

Q9. Application Areas of Numpy?

Q10. What topics will be covered as the part of this numpy?

Q1. What is the need of NumPy?

By using Python we can perform basic mathematical operations.

```
a=10
b=20
a+b
math.sqrt(4)
etc
```

As the part of Data Science,Machine Learning,Deep Learning it is the very common requirement to perform complex mathematical operations like

1. **Creation of Arrays/Matrices**
2. **Perform several operations on arrays/matrices**
3. **perform integral calculus operations**
4. **Solving Differential equations**
5. **Statistics related operations etc**

To perform these complex mathematical operations, python does not contain any inbuilt library.

To perform these complex operations we require a library which is nothing but numpy.

Note:

Just because of these exta libraries like numpy,pandas,matplotlib,skleanr etc, Python is recommended language for Data Science, Machine Learning and Deep Learning etc

scatterplot,heatmap seaborn,sckitlearn,tensorflow

Q2. What is Numpy and History?

NumPy stands for Numerical Python Library.

It is the fundamental python library to perform complex numerical operations.

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.

Numpy is freeware and open source library.

Q3. In Which language Numpy was written?

Numpy was written in Python and C languages.

As most of numpy written in C, performance wise Numpy is good(speed is more).

Because of high speed, numpy is best choice for ML algorithms than traditional python's in built data structures like List.

Q4. What is the purpose of Numpy?

Q6. What are various Features of Numpy?

1. Numpy is superfast because it is written in C language.

2. 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.

3. Numpy has vectorization feature which improves performance while iterating elements.

Q7. What is nd array in Numpy?

In Numpy, we can hold data by using Array Data Structure.

array: An indexed collection of homogeneous elements.

The arrays which are created by using numpy are called nd arrays.

nd array --->N-Dimensional Array or Numpy Array

1-D Array is known as Vector

2-D Array is known as Matrix

This nd array is most commonly used in Data Science Libraries like pandas, scikit learn etc

Numpy library contains several functions to create nd arrays and to perform several required operations.

Q8. What is the advantage of numpy array over Python's in built List?

Performance is very high

Q9. Application Areas of Numpy?

1. To perform lineral algebra functions

2. To perform linear regression

3. To perform logistic regression

4. Deep Neural Networks

5. K-means clustering

6. Control Systems

7. Operational Reasearch

etc

Numpy is the fundamental and compulsory required library for DataScience, Machine Learning, Deep Learning etc

Q10. What topics will be covered as the part of this numpy?

1. Creation of Numpy Array

2. Array Operations

3. Array Attributes

4. Array Indexing and Slicing

5. Broadcasting

6. Iterating Over Array

7. Binary Condition

8. Copy and View

9. Sort and Search

10. Statistics related functions

11. Linear Algebra functions

etc

Q5. 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

Summary:

----

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.

Numpy vs Traditional Python code (Performance Test)

The performance of Numpy is too good when compared with traditional python code.

sample code:

```
import numpy as np
from datetime import datetime
a = np.array([10,20,30])
b = np.array([1,2,3])
print(type(a)) #<class 'numpy.ndarray'>

# Traditional Python code to find dot product: 10X1+20X2+30X3=140
def dot_product(a,b):
        result = 0
        for m,n in zip(a,b):
                result = result + m*n
        return result

before = datetime.now()
```

```
for i in range(1000000):
        dot_product(a,b)
after = datetime.now()
print('The Time Taken:', after-before)

# NumPy Library code to find dot product
before = datetime.now()
for i in range(1000000):
        np.dot(a,b)
after = datetime.now()
print('The Time Taken:', after-before)

D:\durgaclasses>py test.py
<class 'numpy.ndarray'>
The Time Taken: 0:00:02.496784
The Time Taken: 0:00:02.054956

D:\durgaclasses>py test.py
<class 'numpy.ndarray'>
The Time Taken: 0:00:02.482245
The Time Taken: 0:00:01.972443

D:\durgaclasses>py test.py
<class 'numpy.ndarray'>
The Time Taken: 0:00:02.464734
The Time Taken: 0:00:01.969930
```

What is an Array?

An indexed collection of homogeneous data elements is nothing but array.

It is the most commonly used concept in programming languages like C/C++/Java etc

Bydefault arrays concept is not available in python, instead we can use List.

(But make sure list and array both are not same)

But in Python, we can create arrays in the following 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 int type 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 to use because much library support is not available.

2. By using numpy module:

```
import numpy
a = numpy.array([10,20,30])
print(type(a))
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 List vs Numpy Arrays:

----

1. Similarities

2. Differences

1. Similarities between Python List and Numpy Array:

----

1. Both can be used to store data.

2. The order will be presered 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.

2. Differences between Python List and Numpy Array:

----

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.

```
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.

Creation of Numpy Arrays:

Numpy library contains several functions to create ndarray based on our requirement.

The following are few of such functions:

1. array()

2. arange()

3. linspace()

4. zeros()

5. ones()

6. full()

7. eye()

8. identity()

9. empty()

10. random library functions

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

1. array():

To create an array for the given list or tuple.

```
>>> import numpy as np
>>> help(np.array)
Help on built-in function array in module numpy:

array(...)
    array(object, dtype=None, *, copy=True, order='K', subok=False, ndmin=0,
```

like=None)

Create an array.

eg-1: To create 1-Dimensional Array for the given list

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

eg-2: To create 2-Dimensional Array for the given nested list

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

Note: By using ndim attribute, we can find dimension of the array.

eg-3: Create 1-Dimensional array from the given tuple.

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

Q. How to find type of elements in nd array?

By using dtype attribute.

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

If the list contains different types of elements:

While creating nd array, lower type elements will be promoted to higher type automatically(upcasting).

```
>>> a = np.array([10,20.5,30])
>>> a
array([10. , 20.5, 30. ])
>>> type(a)
<class 'numpy.ndarray'>
>>> a.dtype
dtype('float64')
```

How to create array of particular data type:

We have to use dtype argument explicitly.

eg-1:

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

eg-2:

```
>>> a = np.array([10,20,30],dtype=complex)
>>> a
array([10.+0.j, 20.+0.j, 30.+0.j])
>>> a.dtype
dtype('complex128')
```

eg-3: To create bool array

zero is treated as False

non-zero treated as True

empty string treated as False

non-empty string treated as True

eg-4:

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

eg-4a:

```
>>> a = np.array(['A','B',''],dtype=bool)
>>> a
array([ True,  True, False])
```

eg-5:

```
>>> a = np.array([10,20,30,0],dtype=str)
>>> a
array(['10', '20', '30', '0'], dtype='<U2')
>>> a.dtype
dtype('<U2')
```

How to create object type array:

If the elements are different types then we can create object array.

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

If we are not specifying dtype=object

```
>>> a = np.array([10,10.5,True,'durga'])
>>> a
array(['10', '10.5', 'True', 'durga'], dtype='<U32')
```

2. arange():

It's functionality exactly same as python's inbuilt function: range()

range(n): represents a range of values from 0 to n-1

range(m,n): represents a range of values from m to n-1

range(m,n,step)---> represents a range of values from m to n-1 with intervals of step.

range(10) --->0,1,2,....9

range(1,10) --->1,2,3,....9

range(1,10,2) --->1,3,5,7,9

The default value for step is 1

Syntax:

----

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

Return evenly spaced values within a given interval.

eg-1:

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

eg-2:

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

eg-3:

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

eg-4:

```
>>> a = np.arange(2,11,2,dtype=float)
>>> a
array([ 2.,  4.,  6.,  8., 10.])
```

5. linspace():

Syntax:

linspace(start, stop, num=50, endpoint=True, retstep=False, dtype=None, axis=0)

Return evenly spaced numbers over a specified interval.

Returns 'num' evenly spaced samples, calculated over the interval ['start', 'stop'].

eg-1: To get 50 values between 0 and 1

```
>>> a = np.linspace(0,1)
>>> a
array([0.        , 0.02040816, 0.04081633, 0.06122449, 0.08163265,
       0.10204082, 0.12244898, 0.14285714, 0.16326531, 0.18367347,
```

0.20408163, 0.2244898 , 0.24489796, 0.26530612, 0.28571429,

0.30612245, 0.32653061, 0.34693878, 0.36734694, 0.3877551 ,

0.40816327, 0.42857143, 0.44897959, 0.46938776, 0.48979592,

0.51020408, 0.53061224, 0.55102041, 0.57142857, 0.59183673,

0.6122449 , 0.63265306, 0.65306122, 0.67346939, 0.69387755,

0.71428571, 0.73469388, 0.75510204, 0.7755102 , 0.79591837,

0.81632653, 0.83673469, 0.85714286, 0.87755102, 0.89795918,

0.91836735, 0.93877551, 0.95918367, 0.97959184, 1.])

eg-2: >>> a = np.linspace(0,1,5)

>>> a

array([0. , 0.25, 0.5 , 0.75, 1.])

eg-3:

>>> a = np.linspace(0,360,4)

>>> a

array([0., 120., 240., 360.])

eg-4:

>>> np.linspace(0,10,5,dtype=float)

array([0. , 2.5, 5. , 7.5, 10.])

>>> np.linspace(0,10,5,dtype=int)

array([0, 2, 5, 7, 10])

arange() vs linspace()

----

arange(): elements will be considered in the given range based on step value.

linspace(): The specified number of elements will be considered in the given range

eg:

np.arange(0,10,2) # [0,2,4,6,8]

np.linspace(0,10,2)#[0.0, 10.0]

4. zeros():

Syntax:

----

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

Return a new array of given shape and type, filled with zeros.

1-D: Array of elements

2-D: Array of arrays

3-D: Array of array of arrays(array of 2d arrays)

eg-1:

```
>>> np.zeros(4)
array([0., 0., 0., 0.])
```

eg-2:

```
>>> np.zeros((4,))
array([0., 0., 0., 0.])
```

eg-3:

```
>>> np.zeros((2,3))
array([[0., 0., 0.],
       [0., 0., 0.]])
```

eg-4:

```
>>> np.zeros((2,3),dtype=int)
array([[0, 0, 0],
       [0, 0, 0]])
```

eg-5:

```
>>> np.zeros((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]]])
```

eg-6:

```
>>> a = np.zeros((5,),dtype=int)
>>> a
array([0, 0, 0, 0, 0])
>>> a.shape
(5,)
>>> a.ndim
1
>>> a.size
5
```

eg-7:

```
>>> a = np.zeros((2,5),dtype=int)
>>> a
array([[0, 0, 0, 0, 0],
```

[0, 0, 0, 0, 0]])

>>> a.shape

(2, 5)

>>> a.ndim

2

>>> a.size

10

Note:

shape: it is a tuple of integers indicating number of elements in each dimension

eg: (2,3,4)

It is 3-dimesional array where first dimension contains 2 elements and second dimension contains 3 elements and 3rd dimension contains 4 elements.

Total size of this array: 24

size: The total number of elements present in the given array(2*3*4).

Q. An N-D array will contain elements in nth dimension

Yes.

5. ones():

Syntax:

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

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

eg-1:

>>> np.ones(3)

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

eg-2:

```
>>> np.ones((3,))
array([1., 1., 1.])
```

eg-3:

```
>>> a = np.ones((3,2),dtype=int)
>>> a
array([[1, 1],
       [1, 1],
       [1, 1]])
>>> a.shape
(3, 2)
>>> a.size
6
>>> a.ndim
2
```

6. full():

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

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

It is exactly same as zeros() and ones() functions except that we can fill with our required value(fill_value)

```
>>> np.full((5,),7)
array([7, 7, 7, 7, 7])
>>> np.full((3,5),6,dtype=float)
```

```
array([[6., 6., 6., 6., 6.],
       [6., 6., 6., 6., 6.],
       [6., 6., 6., 6., 6.]])
>>> np.full((2,3),[10,20,30])
array([[10, 20, 30],
       [10, 20, 30]])
```

7. eye()

Syntax:

```
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.

    Parameters
    ----------
    N : int
      Number of rows in the output.
    M : int, optional
      Number of columns in the output. If None, defaults to `N`.
    k : int, optional
      Index of the diagonal: 0 (the default) refers to the main diagonal,
      a positive value refers to an upper diagonal, and a negative value
      to a lower diagonal.
    dtype : data-type, optional
      Data-type of the returned array.
```

eg-1:

```
>>> np.eye(3)
array([[1., 0., 0.],
```

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

eg-2:

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

eg-3:

```
>>> np.eye(4,5)
array([[1., 0., 0., 0., 0.],
       [0., 1., 0., 0., 0.],
       [0., 0., 1., 0., 0.],
       [0., 0., 0., 1., 0.]])
```

eg-4:

```
>>> np.eye(4,5,-1)
array([[0., 0., 0., 0., 0.],
       [1., 0., 0., 0., 0.],
       [0., 1., 0., 0., 0.],
       [0., 0., 1., 0., 0.]])
```

eg-5:

```
>>> np.eye(4,5,-2)
array([[0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.],
       [1., 0., 0., 0., 0.],
       [0., 1., 0., 0., 0.]])
```

eg-6:

```
>>> np.eye(4,5,1)
array([[0., 1., 0., 0., 0.],
       [0., 0., 1., 0., 0.],
       [0., 0., 0., 1., 0.],
       [0., 0., 0., 0., 1.]])
```

eg-7:

```
>>> np.eye(4,5,-10)
array([[0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.],
       [0., 0., 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

8. identity():

Syntax:

---

identity(n, dtype=None, *, like=None)

Return the identity array.

The identity array is a square array with ones on

the main diagonal.

Parameters

---

n : int

Number of rows (and columns) in `n` x `n` output.

dtype : data-type, optional

Data-type of the output. Defaults to ``float``.

eg-1:

```
>>> np.identity(3)
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
```

eg-2:

```
>>> np.identity(5,dtype=int)
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]])

eg-3:

>>> np.identity(5,4,dtype=int)

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

TypeError: identity() got multiple values for argument 'dtype'

Note: eye() function is generalized function where as identity() function is most specific function.

identity() function is a special case of eye() function where

1. It should be square matrix(2D array with equal number of rows and columns)

2. Main diagonal should contains ones

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

Revision:

----

eye()--->to generate 2-D identity matrix

--->The number of rows and columns need not be same

--->Instead of main diagonal, we can specify diagonal which has to contains 1s.

np.eye(2,k=0)

np.eye(3,2,k=-1)

----

np.identity()

---> to generate 2-D identity matrix

--->The number of rows and columns must be same

--->only main diagonal should contain 1s

np.identity(3)

Q. Sir in eye if k=0 always main diagonal will have 1's?

Yes and 0 is the default value for k

9. empty():

----

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

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

Returns

----

out : ndarray

Array of uninitialized (arbitrary) data of the given shape, dtype, and

order. Object arrays will be initialized to None.

eg:

```
>>> np.empty((2,3))
array([[2.33419537e-312, 8.48798317e-313, 1.27319747e-312],
       [1.69759663e-312, 2.12199580e-313, 6.36598737e-313]])
```

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
```

10. Array Creation by using random library:

Numpy's random library is almost same as Python's inbuilt random library, which can be used to generate random numbers. We can create array with those random values as array elements.

1. randint():

To generate random int values.

Syntax:

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

- Return random integers from 'low' (inclusive) to 'high' (exclusive).

size : int or tuple of ints, optional

Output shape. If the given shape is, e.g., (m, n, k), then

m * n * k samples are drawn. Default is None, in which case a

single value is returned.

eg-1:

np.random.randint(10,20)

It will generate a random int number which is >=10 but <20. (i.e from 10 to 19)

eg-2:

np.random.randint(20)

It will generate a random number which is in between 0 to 19.

```
>>> np.random.randint(10,20)
10
>>> np.random.randint(10,20)
18
>>> np.random.randint(10,20)
12
>>> np.random.randint(10,20)
10
>>> np.random.randint(10,20)
12
>>> np.random.randint(10,20)
10
>>> np.random.randint(10,20)
15
>>> np.random.randint(10,20)
13
>>> np.random.randint(10,20)
12
>>> np.random.randint(10,20)
17
```

```
>>> np.random.randint(10,20)
16
>>> np.random.randint(10)
2
>>> np.random.randint(10)
4
>>> np.random.randint(10)
5
>>> np.random.randint(10)
1
>>> np.random.randint(10)
2
>>> np.random.randint(10)
9
>>> np.random.randint(10)
4
>>> np.random.randint(10)
8
>>> np.random.randint(10)
4
>>> np.random.randint(10)
6
>>> np.random.randint(10)
0
>>> np.random.randint(10)
```

eg-1: Create nd array of one dimension of 10 size with random values from 1 to 8.

```
>>> np.random.randint(1,9,size=10)
```

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


>>> np.random.randint(1,9,10)
array([4, 7, 4, 7, 6, 7, 4, 7, 3, 4])
```

eg-2: To create 2D array with shape(3,5)

```
>>> np.random.randint(100,size=(3,5))
array([[45, 15, 41, 26, 26],
       [10, 20, 40, 16, 19],
       [44, 11, 67,  8, 35]])
```

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

```
>>> np.random.randint(100,size=(2,3,4))
array([[[33, 88, 25, 84],
        [85, 18, 55, 45],
        [76, 57, 45, 80]],

       [[ 9, 65, 65, 27],
        [75, 61,  0,  6],
        [52, 28, 81, 58]]])
```

Note: Here dtype can be any int type like int32 or int64 etc, but we cannot take any other types like float.

```
>>> a = np.random.randint(1,11,7,dtype=int)
>>> a
array([ 9,  9,  1, 10,  8,  5,  5])
>>> a.dtype
dtype('int32')
```

```
>>> a = np.random.randint(1,11,7,dtype='int8')
>>> a.dtype
dtype('int8')
>>> a = np.random.randint(1,11,7,dtype='int16')
>>> a.dtype
dtype('int16')



>>> a = np.random.randint(1,11,7,dtype='int32')
>>> a
array([ 2,  1,  7, 10,  5,  6,  2])
>>> a.dtype
dtype('int32')

>>> a = np.random.randint(1,11,7,dtype='int64')
>>> a.dtype
dtype('int64')

>>> a = np.random.randint(1,11,7,dtype='float')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "mtrand.pyx", line 763, in numpy.random.mtrand.RandomState.randint
TypeError: Unsupported dtype dtype('float64') for randint

How to convert from one array type to another array type:
-----------------------------------------------------------
>>> a = np.random.randint(1,11,7,dtype='int16')
```

```
>>> a.dtype
dtype('int16')
>>> a
array([1, 1, 1, 3, 4, 5, 7], dtype=int16)
>>> b = a.astype('float')
>>> b
array([1., 1., 1., 3., 4., 5., 7.])
>>> 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() function

---

rand(d0, d1, ..., dn)

-Random values in a given shape.

-Create an array of the given shape and populate it with random samples from a uniform distribution over [0, 1). 0 inclusive but 1 exclusive.

Parameters

---

d0, d1, ..., dn : int, optional

The dimensions of the returned array, must be non-negative.

If no argument is given a single float value will be generated.

eg-1:

```
>>> np.random.rand()
0.054405190205850884
>>> np.random.rand()
0.040647897775599295
```

eg-2:

```
>>> np.random.rand(3)
array([0.83912901, 0.64730918, 0.65829491])
```

eg-3:

```
>>> np.random.rand(2,3)
array([[0.88186185, 0.19833909, 0.17236504],
       [0.01180205, 0.08002359, 0.50064927]])
```

eg-4:

```
>>> np.random.rand(2,3,5)
array([[[0.70663592, 0.23348426, 0.15253474, 0.10795603, 0.39937198],
        [0.17003214, 0.41018137, 0.02026827, 0.323256  , 0.01589707],
        [0.92207686, 0.14700846, 0.78516395, 0.02655265, 0.0697182 ]],

       [[0.52117078, 0.59200017, 0.9239475 , 0.11608647, 0.58055829],
        [0.46940265, 0.79212916, 0.50972441, 0.37749165, 0.27939628],
        [0.51238925, 0.87572578, 0.04739316, 0.17827908, 0.53442166]]])
```

3. uniform():

Syntax:

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

Draw samples from a uniform distribution.

Samples are uniformly distributed over the half-open interval ''[low, high)'' (includes low, but excludes high).

eg-1: If we are not passing any argument, it simply acts as rand() function.

```
>>> np.random.uniform()
0.9439394959018282
>>> np.random.uniform()
0.7970971752164758
>>> np.random.uniform()
0.5068985659487931
>>> np.random.uniform()
0.20453164527515644
>>> np.random.uniform()
0.5175842954748464
```

eg-2:

```
>>> np.random.uniform(10,20)
12.588199652498417
>>> np.random.uniform(10,20)
16.44176578692628
>>> np.random.uniform(10,20)
12.156096979661884
>>> np.random.uniform(10,20)
19.44994224690347
```

eg-3:

```
>>> np.random.uniform(10,20,size=10)
```

array([12.8460974 , 12.5909643 , 15.62314161, 17.78570169, 14.92958804,

10.89612554, 19.71291354, 18.79248641, 13.03500299, 14.16827282])

eg-4:

>>> np.random.uniform(10,20,size=(2,3))

array([[10.31451286, 11.07349691, 11.46088957],

[19.04364601, 13.81225102, 12.61413272]])

eg-5:

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()

randn()

Syntax:

randn(d0, d1, ..., dn)

Return a sample (or samples) from the "standard normal" distribution of mean 0 and variance 1.

Parameters

----

d0, d1, ..., dn : int, optional

The dimensions of the returned array, must be non-negative.

If no argument is given a single Python float is returned.

Returns

Z : ndarray or float

A ``(d0, d1, ..., dn)``-shaped array of floating-point samples from

the standard normal distribution, or a single such float if

no parameters were supplied.

```
>>> np.random.randn()
1.0156920146285195
>>> np.random.randn(3)
array([ 0.02522907,  0.38010197, -0.27503773])
>>> np.random.randn(3,2)
array([[ 0.9393751 ,  0.58826297],
       [ 0.00924243,  1.21003746],
       [ 1.0109597 , -1.6088758 ]])
>>> np.random.randn(2,3,4)
array([[[ 0.67624207, -1.46274234,  0.00732033, -0.33019154],
        [ 1.80526216,  0.71892504, -0.13447181,  2.0164551 ],
        [-1.40540834, -0.22872869,  0.60825913,  0.63015516]],

       [[-0.32339663, -2.09548099, -0.09854932,  0.37776156],
        [-0.16263054,  0.13338616,  0.27997952, -0.49005018],
        [-0.34096277,  0.63203582, -0.0425311 , -1.08976535]]])
```

3. normal():

----

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

Draw random samples from a normal (Gaussian) distribution.

Parameters

----

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(varience)

size : int or tuple of ints, optional

Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.

If we are not passing any value, then a single float value will be returned.

eg-1: If we are not passing any argument, then it acts as randn() function.

```
>>> np.random.normal()
0.3733151175386248
>>> np.random.normal()
-0.9073500065910385
>>> np.random.normal()
-0.965320002780851
```

eg-2:

```
>>> np.random.normal(10,4)
12.597959123744054
>>> np.random.normal(10,4)
12.4328685244566
>>> np.random.normal(10,4)
12.264023606692891
```

>>> np.random.normal(10,4)

10.736425424120322

eg-3:

```
s = np.random.normal(10,4,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()
```

Note:

1.rand()--->uniform distributed float values from [0,1).

2.uniform()--->uniform distributed float values in the given range.

3. randn()--->normal distributed float values with mean 0 and variance 1.

4. normal()--->normal distributed float values with specified mean and variance.

3. shuffle():

Syntax:

------

shuffle(x)

Modify a sequence in-place by shuffling its contents.

Parameters

----

x : ndarray or MutableSequence

The array, list or mutable sequence to be shuffled.

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

Returns

In the existing array only, the modification will be happend. Hence it returns None.

eg-1:

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

eg-2:

In 2-D array, only rows will be shuffled but not internal elements.

```
>>> a = np.random.randint(1,101,size=(6,5))
>>> a
array([[ 89,   8,  69,  29,  48],
       [ 70, 100,  41,  99,  56],
       [ 70,  41,  62,  95,  86],
       [ 49,   7,  27,  76,  87],
       [ 44,  27,  44,   2,  57],
```

[43, 5, 34, 77, 26]])

>>> np.random.shuffle(a)

>>> a

array([[89, 8, 69, 29, 48],

[70, 41, 62, 95, 86],

[43, 5, 34, 77, 26],

[44, 27, 44, 2, 57],

[49, 7, 27, 76, 87],

[70, 100, 41, 99, 56]])

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.

matplot lib graphs:

----

pip install numpy

pip install matplotlib

Display the histogram of the samples, along with the

probability density function:

```
s = np.random.uniform(-1,0,1000)
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()
```

```
s = np.random.rand(100000)
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()
```

```
s = np.random.rand(5)
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()
--------------------------------------------------
s = np.random.randn(100000)
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()
----------------------------------------------------------
s = np.random.uniform(20,30,size=1000000)
import matplotlib.pyplot as plt
count, bins, ignored = plt.hist(s, 15, density=True)
plt.plot(bin
```

Summary:

----

1. array()

2. arange()

3. linspace()

4. zeros()

5. ones()

6. full()

7. eye()

8. identity()

9. empty()

10. random library functions

- **1. randint()**
- **2. rand()**

3. uniform()

4. randn()

5. normal()

6. shuffle()

Array Attributes:

The following are various array attributes.

1. ndim: To get dimension of the array

2. shape: To get shape of the array

3. size: To get size of the array which is nothing but number of elements.

4. dtype: To get data type of array elements

5. itemsize: The size of each array element in bytes

eg-1:

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

eg-2:

```
>>> a = np.array([[10,20,30],[40,50,60]],dtype=float)
>>> a
array([[10., 20., 30.],
       [40., 50., 60.]])
>>> a.ndim
2
>>> a.shape
(2, 3)
>>> a.size
6
>>> a.dtype
dtype('float64')
>>> a.itemsize
8
```

Numpy Data Types:

In Python we have data types like int,float,complex,bool,str etc.

But, Numpy has some extra data types in addition to these types. We can represent data types by using a single character 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 using 8 bits.

The range of values: -128 to 127

int16

The value will be represented by using 16 bits.

The range of values: -32768 to 32767

int32

The value will be represented by using 32 bits.

The range of values: -2147483648 to 2147483647

int64

The value will be represented by using 64 bits.

The range of values: -9223372036854775808 to 9223372036854775807

Note:

1. By default int means int32

2. By default float means float64

Note: int8 type array requires less memory than int32 type array.

```
>>> a = np.array([10,20,30,40])
>>> import sys
>>> sys.getsizeof(a)
120

>>> a = np.array([10,20,30,40],dtype='int8')
>>> sys.getsizeof(a)
108
```

How to check the data type of an array?

By using dtype attribute.

eg:

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


>>> a = np.array([10.5,20.6,30])
>>> a.dtype
dtype('float64')
```

How to create array with required data type?

By using dtype argument

```
>>> a = np.array([10,20,30,40],dtype='i8')
>>> a.dtype
```

```
dtype('int64')
>>> a = np.array([10,20,30,40],dtype='int8')
>>> a.dtype
dtype('int8')
>>> a = np.array([10,20,30,40],dtype='i')
>>> a.dtype
dtype('int32')
```

Note: If the array element unable to convert into our specified type, then we will get error.

```
>>> 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?

In general we can use astype() function.

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

Note: By using the corresponding data type function also, we can convert the type of array.

eg-1:

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

eg-2:

```
>>> a = np.array([10,20,30,0,40])
>>> a.dtype
dtype('int32')
>>> x = np.bool_(a)
>>> x
array([ True,  True,  True, False,  True])
>>> x.dtype
dtype('bool')
```

How to get/access elements of Numpy Array:

There are 2 ways available to to get/access elements of Numpy Array

1. Indexing

2. Slicing

1. Indexing:

----

1. By using index, we can access single element of the array.

2. Numpy follows zero based indexing. ie index of first element is always zero.

3. Numpy supports both positive indexing and negative indexing.

Syntax:

a[index] --->Returns element present at specified index.

Accessing elements from 1D array:

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

Diagram

```
>>> a[0]
10
>>> a[2]
30
>>> a[-1]
50
>>> a[-2]
40
```

Note: If we are trying to access array element with out of range index, then we will get IndexError.

eg:

```
>>> 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
>>> a[-7]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: index -7 is out of bounds for axis 0 with size 5
```

Accessing elements of 2D array:

Syntax:

a[i][j]

i-->means row index

j-->means column index

eg:

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

Diagram

1. To access 50:

---

a[1][1]

a[-1][-2]

a[1][-2]

a[-1][1]

2. To access 30:

----

a[0][2]

a[-2][-1]

a[0][-1]

a[-2][2]

Q. Which elements returned by the following expressions?

A. a[0][1] --->20

B. a[-2][0] ---->10

C. a[-1][-2] --->50

D. a[-3][-1] --->IndexError

E. a[0][4]---->IndexError

Accessing elements of 3D array:

3D array means array of 2D arrays.

Syntax:

a[i][j][k]

i represents which 2-Dimensional array

j represents row number in that 2-D array

k represents column number in that 2-D array

eg:

a[0][1][2]

0 index 2-D array

In that 2-D array, 1 index row number

In that 2-D array, 2-index column number

eg-1:

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

       [[10, 11, 12],
        [13, 14, 15],
        [16, 17, 18]]])
>>> a.shape
(2, 3, 3)
```

2--->Number of 2-D arrays

3 ---> in every 2-D arrays, 3 number of rows

3 ---> in every 2-D array, 3 number of columns

eg-2:

-----

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)

2--->2-D arrays

3 --->3 Rows in every 2-D array

4 ---> 4 Columns in every 2-D Array

shape=(2,3,4)

It is 3D array which contains two 2D arrays. Each 2D array contains 3 rows and 4 columns. Total 24 elements are there.

a[i][j][k]

i--->which 2D array

j--->In that 2D array, Row index

k --->In that 2D array, column index

eg:

```
>>> 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
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]]])
```

Diagram

To access 18 :

----

a[1][1][1]

a[-1][-2][-3]

a[1][-2][-3]

a[1][1][-3]

a[-1][1][1]

a[-1][-2][1]

To access 7:

----

a[0][1][2]

a[-2][-2][-2]

Accessing elements of 4-D array:

4D array means an array of 3-D arrays.

Diagram

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

i-->which 3d array

j-->in that 3d array, which 2d array

k--->in that 2d array which row

l --->in that 2d array which column

Accessing elements of array by using slice operator:

Slice means part of the object.

Slice operator on Python List:

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

Diagram

Syntax-1:

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

The default value for begin: 0

The default value for end: len(l)

```
>>> l[1:6] ---->It returns elements from 1st index to 5th index
[20, 30, 40, 50, 60]
>>> l[-6:-2] ---->It returns elements from -6th index to -3 index
[20, 30, 40, 50]
>>> l[:4] ----->It returns elements from 0 index to 3 index
[10, 20, 30, 40]
>>> l[2:]--->It returns elements from 2nd index to last element
[30, 40, 50, 60, 70]
>>> l[4:1]
[]
>>> l[4:10000]
[50, 60, 70]
```

>>> l[-10:]

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

>>>l[:]

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

Note: Slice operator won't raise IndexError

Syntax-2:

l[begin:end:step]

---It returns elements from begin index to end-1 index based on step value

Step is optional and default value 1.

l[1:6:1]--->It returns elements from 1st index to 5th index and every element will be considered.

l[1:6:2]--->It returns elements from 1st index to 5th index and every 2nd element will be considered.

l[1:6:3]--->It returns elements from 1st index to 5th index and every 3rd element will be considered.

>>> l[1:6:1]

[20, 30, 40, 50, 60]

>>> l[1:6:2]

[20, 40, 60]

>>> l[1:6:3]

[20, 50]

Rules of slice operator:

----

1. All 3 parameters of slice operator are optional.

2.For begin,end,step attributes, we can take both positive and negative values.

But for step we cannot take zero.

```
>>> l[1:6:0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: slice step cannot be zero
```

3. Based on step value, we can decide whether we have to consider elements in either forward or backward direction.

If step value is positive, We have to consider elements from begin to end-1 in forward direction.

If step value is negative, We have to consider elements from begin to end+1 in backward direction.

4. In forward direction, if end value is 0 then result is always empty.

5. In backward direction, if end value is -1 then result is always empty.

eg:

l[5:1:-1] --->Return elements from 5th index to 2nd index in backward direction

l[5:1:-2] --->Return elements from 5th index to 2nd index in backward direction,every second element we have to consider.

l[4:-5:1]--->from 4th index to -6 in forward direction

```
>>> l[5:1:-1]
[60, 50, 40, 30]
>>>
>>> l[5:1:-2]
```

[60, 40]

>>> l[4:-5:1]

[]

l[-2:-6:-1]---> from -2 to -5 in backward direction

l[5:1:-1]--->from 5th index to 2nd index in backward direction

l[-1::-2]--->from -1 index to beginning of the list, every 2nd element in backward direction

l[::]--->All elements in forward direction

l[::-1] ---> All elements in backward direction

>>> l[-2:-6:-1]

[60, 50, 40, 30]

>>> l[5:1:-1]

[60, 50, 40, 30]

>>> l[-1::-2]

[70, 50, 30, 10]

>>> l[::]

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

>>> l[::-1]

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

l[-1:-1:-1]--->Empty

Slice Operator on 1-D Numpy Array:

The rules are exactly same as Python's in built slice operator.

Syntax: a[begin:end:step]

Rules:

------

1. All 3 parameters of slice operator are optional.

2.For begin,end,step attributes, we can take both positive and negative values.

But for step we cannot take zero.

```
>>> l[1:6:0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: slice step cannot be zero
```

3. Based on step value, we can decide whether we have to consider elements in either forward or backward direction.

If step value is positive, We have to consider elements from begin to end-1 in forward direction.

If step value is negative, We have to consider elements from begin to end+1 in backward direction.

4. In forward direction, if end value is 0 then result is always empty.

5. In backward direction, if end value is -1 then result is always empty.

eg:

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

Diagram

```
>>> a[2:5]
array([30, 40, 50])
>>> a[:5]
```

```
array([10, 20, 30, 40, 50])
>>> a[2:]
array([30, 40, 50, 60, 70])
>>> a[::2]
array([10, 30, 50, 70])
```

Slice Operator on 2-D Numpy Array:

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

Diagram

Syntax:

```
arrayname[row,column]
arrayname[begin:end:step,begin:end:step]
```

The first slice operator talks about rows and second slice operator talks about columns.

eg-1:

```
>>> a[:,0:1]
array([[10],
       [30],
       [50]])
```

```
>>> a[:,:1]
array([[10],
       [30],
       [50]])
```

Note: For both row and column we should use slice operator only. If we use index directly we may get 1-D array as the result, but not 2-D array.

```
>>> a[:,0]
array([10, 30, 50])
```

eg-2:

```
>>> a[0:2,:]
array([[10, 20],
       [30, 40]])
>>> a[:2,:]
array([[10, 20],
       [30, 40]])
```

eg-3:

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

Q1. Difference bet :2 & ::2?

:2 --->2 acts as end index

::2 --->2 acts as step value

case study:

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

Diagram

eg-1:

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

eg-2:

```
>>> a[0::3,:]
array([[ 1,  2,  3,  4],
       [13, 14, 15, 16]])
>>> a[::3,:]
array([[ 1,  2,  3,  4],
       [13, 14, 15, 16]])
```

eg-3:

```
>>> a[:3,::2]
array([[ 1,  3],
```

```
     [ 5,  7],
     [ 9, 11]])
```

eg-4:

```
>>> a[1:3,1:3]
array([[ 6,  7],
     [10, 11]])
```

eg-5:

```
>>> a[::3,::3]
array([[ 1,  4],
     [13, 16]])
```

Q. How we can select only arbitrary elements?

We cannot use slice operator because there is no order. But we can select arbitraty elements by using advanced indexing.

Slice Operator on 3D 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
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]]])
```

>>> a.shape

(2, 3, 4)

Diagram

Syntax:

a[i,j,k]

i--->For which 2D arrays

j--->In those 2D arrays, which rows

k --->In those 2D arrays, which columns

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

eg-1:

```
>>> a[0:1,0:2,0:2]
array([[[1, 2],
        [5, 6]]])
>>> a[:1,:2,:2]
array([[[1, 2],
        [5, 6]]])
```

eg-2:

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

       [[13],
```

```
       [17],
       [21]]])
>>> a[:,:,:1]
array([[[ 1],
       [ 5],
       [ 9]],

      [[13],
       [17],
       [21]]])
```

eg-3:

```
>>> a[:,1:,1:3]
array([[[ 6,  7],
       [10, 11]],

      [[18, 19],
       [22, 23]]])


>>> a[:,1:3,1:3]
array([[[ 6,  7],
       [10, 11]],

      [[18, 19],
       [22, 23]]])
```

eg-4:

```
>>> a[:,::2,::3]
array([[[ 1,  4],
        [ 9, 12]],

       [[13, 16],
        [21, 24]]])
```

eg-5:

```
>>> a[:,::2,::]
array([[[ 1,  2,  3,  4],
        [ 9, 10, 11, 12]],

       [[13, 14, 15, 16],
        [21, 22, 23, 24]]])
```

Advanced Indexing:

By using index we can access only one element at a time.

Syntax: a[i], a[i][j], a[i][j][k]

By using Slice operator, we can access multiple elements but should be in the order.

Syntax:

a[begin:end:step],

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

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

If we want to access multiple elements which are not in order(arbitrary elements) or if we want to access elements based on some condition then we cannot use basic indexing and slicing operators. For this requirement we should go for Advanced Indexing.

Accessing multiple arbitrary elements:

Accessing elements of 1-D array:

Syntax: array[x]

--->x can be either nd array or list, which represents required indexes.

eg-1:

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

Diagram

To access 10,30,80:

1st way:

---

create nd array with required indexes and then by passing that array as argument we can access corresponding elements.

```
>>> indexes = np.array([0,2,7])
>>> a[indexes]
array([10, 30, 80])
```

2nd way:

---

create list with required indexes and then by passing that list as argument we can access corresponding elements.

```
>>> l = [0,2,7]
>>> a[l]
array([10, 30, 80])
>>> a[[0,2,7]]
array([10, 30, 80])
```

Note: Even we can access same element multiple times also.

```
>>> a[[0,2,7,0,0,2,2,7,7]]
array([10, 30, 80, 10, 10, 30, 30, 80, 80])

>>> a[[2,6,-3]]
array([30, 70, 70])
```

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

Diagram

Syntax:

```
a[[row_indexes],[column_indexes]]
```

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

The elements present at (0,0), (1,1), (2,2),(3,3) will be selected.

```
>>> a[[0,1,2,3],[0,1,2,3]]
array([ 1,  6, 11, 16])
9
```

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

The elements present at (1,0),(3,2) will be selected.

```
>>> a[[1,3],[0,2]]
array([ 5, 15])
```

Note: Observation:

```
>>> a[[0,2],[0,1,3]]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: shape mismatch: indexing arrays could not be broadcast together with shapes (2,) (3,)

>>> a[[0,2,3],[0,1]]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: shape mismatch: indexing arrays could not be broadcast together with shapes (3,) (2,)

>>> a[[0,2,3],[1]]
array([ 2, 10, 14])
```

```
>>> a[[0],[0,1,3]]
array([1, 2, 4])
```

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.ndim
3
>>> a.shape
(2, 3, 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]]])
```

Diagram

Syntax:

```
array[[indexes of 2d array],[row indexes],[column indexes]]
```

eg-1:

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

The elements present at (0,0,0),(1,0,1),(1,2,3) will be selected.

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

array([1, 14, 24])

eg-2:

a[[0,1],[1,0],[1,3]]

The elements present at (0,1,1),(1,0,3) will be selected.

>>> a[[0,1],[1,0],[1,3]]

array([6, 16])

eg-3:

>>> a[[-1,-2],[-3,-1],[-1,-3]]

array([16, 10])

Summary:

----

1. To access elements from 1-D Array

a[x] ---->x can be nd array or list which contains indexes

2. To access elements from 2-D array

a[[row_indexes],[column_indexes]]

3. To access elements from 3-D array

a[[indexes of 2-D array],[row_indexes],[column_indexes]]

How to select elements based on some condition:

We can select array elements based on some condition also.

Syntax: array[boolean_array]

It selects all elements where boolean array contains True.

eg-1:

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

To select all elements which are less than 0, First create boolean array.

```
>>> boolean_array = a<0
>>> boolean_array
array([False,  True, False, False,  True,  True, False])
```

By passing this boolean array select array elements

```
>>> a[boolean_array]
array([-5, -3, -1])
```

Short cut: a[a<0]

eg-2: To select all elements which are greater than 10.

```
>>> a[a>10]
array([20, 40, 75])
```

Note: We can use this approach for any dimensional array, But output is always 1-D array.

eg-3:

```
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)
```

Select all elements which are greater than 20.

```
>>> 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[a>20]
array([21, 22, 23, 24])
```

Slicing vs Advanced Indexing:

Case-1: Python's list slicing:

In the case of normal list, slice operator will create a new copy of the object.

If we perform any changes in one copy, those changes won't be reflected in other.

eg:

```
l1=[10,20,30,40]
l2=l1[::]
print(id(l1)) #2793470048384
print(id(l2)) #2793541280832
l2[0]=9999
print(l2) #[9999, 20, 30, 40]
print(l1) #[10, 20, 30, 40]
```

Case-2: Numpy Array Slicing:

But in numpy slicing, a new copy won't be created and just we are getting view of the existing nd array. If we perform any changes in the original array, those changes will be reflected to the sliced copy and even vice-versa also.

```
>>> a1 = np.array([10,20,30,40,50,60,70])
>>> a2 = a1[0:4]
>>> print(a1)
[10 20 30 40 50 60 70]
>>> print(a2)
[10 20 30 40]
>>> a1[0]=7777
>>> print(a1)
[7777   20   30   40   50   60   70]
>>> print(a2)
[7777   20   30   40]
>>> a2[1]=9999
>>> a2
array([7777, 9999,   30,   40])
>>> a1
array([7777, 9999,   30,   40,   50,   60,   70])
```

Case-3: Numpy Array Advanced Indexing:

But in the case of advanced indexing, a new separate copy will be created. If we perform any changes in one copy, those changes won't be reflected in other.

```
>>> a1 = np.array([10,20,30,40,50,60,70])
>>> a2=a1[[0,2,5]]
>>> print(a1)
[10 20 30 40 50 60 70]
```

```
>>> print(a2)
[10 30 60]
>>> a1[0]=8888
>>> print(a1)
[8888   20   30   40   50   60   70]
>>> print(a2)
[10 30 60]
```

Differences between Numpy Slicing and Advanced Indexing:

Slicing	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.
4. Memory and Performance point of view slicing is the best choice.	4. Memory and Performance point of view Advanced Indexing is not best choice.

How to iterate elements of the nd array:

Iteration means getting elements one by one.

We can iterate elements of nd array by using the following ways

1. By using python's loops

2. By using numpy nditer() function

3. By using numpy ndenumerate() function

1. By using python's loops:

---

eg-1: to iterate elements of 1-D array:

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

```
D:\durgaclasses>py test.py
10
20
30
40
50
60
```

eg-2: To iterate elements of 2-D array:

2-D array means array of 1-D arrays.

If we iterate 2-D array by using Python's for loop, we will get 1-D arrays.

Again we have to iterate that 1-D array to get elements.

```
import numpy as np
a = np.array([[10,20,30],[40,50,60]])
print('1-D arrays one by one:')
for x in a:
        print(x)

print('Elements one by one:')
for x in a: # x is 1-D array
        for y in x: # y is element
                print(y)
```

```
D:\durgaclasses>py test.py
1-D arrays one by one:
[10 20 30]
[40 50 60]
Elements one by one:
10
20
30
40
50
60
```

eg-3 To iterate elements of 3-D array:

```
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)


D:\durgaclasses>py test.py
[[[10 20]
  [30 40]]

 [[40 50]
  [60 70]]]
Elements one by one:
10
20
30
40
40
50
60
70
```

Note: To iterate elements of n-D array by using Python, we required n loops, which is not convenient. To overcome this problem we should go for nditer() function.

2. By using Numpy's nditer() function:

---

nditer() function is specially designed function to iterate elements of any n-D array easily without using multiple loops.

But strictly speaking,

nditer is a class present in numpy library.

nditer()---> we are creating an object of nditer class.

eg-1: To iterate elements of 2-D array:

```
import numpy as np
a = np.array([[10,20,30],[40,50,60]])
for x in np.nditer(a): #x is scalar but not n-1 Dimension array
        print(x)
```

```
D:\durgaclasses>py test.py
10
20
30
40
50
60
```

eg-3 To iterate elements of 3-D array:

```
import numpy as np
a = np.array([[[10,20],[30,40]],[[40,50],[60,70]]])
for x in np.nditer(a): #x is scalar but not n-1 Dimension array
```

```
        print(x)
```

```
D:\durgaclasses>py test.py
10
20
30
40
40
50
60
70
```

Note: For any dimension array, one for loop is enough.

eg-4: Iterating elements in sliced array:

```
----------------------------------------
import numpy as np
a = np.array([[10,20,30],[40,50,60],[70,80,90]]) #2-D array
print(a)
for x in np.nditer(a[:,:2]): # all rows, but 0th and 1th columns
        print(x)
```

```
D:\durgaclasses>py test.py
[[10 20 30]
 [40 50 60]
 [70 80 90]]
10
20
```

40

50

70

80

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

While iterating elements of nd array, we can specify our required type. For this, we have to use op_dtypes argument.

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

TypeError: Iterator operand required copying or buffering, but neither copying nor buffering was enabled

Numpy won't change the type of elements in existing array. To store changed type elements, we required temporary storage, which is nothing but buffer. We have to enable that buffer.

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

```
D:\durgaclasses>py test.py
10.0
20.0
30.0
```

40.0

40.0

50.0

60.0

70.0

eg-2:

```
import numpy as np
a = np.array([[[10,20],[30,40]],[[40,50],[60,70]]])
for x in np.nditer(a):
        print(x.dtype) #int32

for x in np.nditer(a,flags=['buffered'],op_dtypes=['int64']):
        print(x.dtype) #int64
```

Normal python's for loop vs nditer()

Table: Normal python's for loop | nditer()

1. To iterate n-dimensional array, n for loops are required.	**1. To iterate n-dimensional 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)**

3. By using numpy ndenumerate() function

By using nditer() we will get elements only but not indexes.

If we want indexes also in addition to elements, then we should use ndenumerate() function.

ndenumerate() function returns multidimensional index iterator which yields pairs of array indexes(coordinates) and values.

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

```
import numpy as np
a = np.array([10,20,30,40,50,60]) #1-D array
for idx, element in np.ndenumerate(a):
        print(f'{element} element present at index:{idx}')
```

```
10 element present at index:(0,)
20 element present at index:(1,)
30 element present at index:(2,)
40 element present at index:(3,)
50 element present at index:(4,)
60 element present at index:(5,)
```

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 idx, element in np.ndenumerate(a):
        print(f'{element} element present at index:{idx}')
```

```
D:\durgaclasses>py test.py
10 element present at index:(0, 0)
20 element present at index:(0, 1)
```

30 element present at index:(0, 2)

40 element present at index:(1, 0)

50 element present at index:(1, 1)

60 element present at index:(1, 2)

70 element present at index:(2, 0)

80 element present at index:(2, 1)

90 element present at index:(2, 2)

eg-2:

```
import numpy as np
a = np.array([[10,20,30],[40,50,60],[70,80,90]])
for idx, element in np.ndenumerate(a):
        print(f'{element+9999} element present at index:{idx}')
```

```
D:\durgaclasses>py test.py
10009 element present at index:(0, 0)
10019 element present at index:(0, 1)
10029 element present at index:(0, 2)
10039 element present at index:(1, 0)
10049 element present at index:(1, 1)
10059 element present at index:(1, 2)
10069 element present at index:(2, 0)
10079 element present at index:(2, 1)
10089 element present at index:(2, 2)
```

Arithmetic Operators :

The following are various arithmetic operators:

+ --->Addition

- --->Subtraction

*** --->Multiplication**

/ --->Division

// ----->Floor Division

% --->Modulo operation/Remainder Operation

**** --->Exponential operation/power operation**

Note:

The result of division operator(/) is always float. But floor division operator(//) can return either integer and float values. If both arguments are of type int, then floor division operator returns int value only. If atleast one argument is float type then it returns float type only.

eg-1:

```
print(3/2) #1.5
print(3//2) #1
print(3.0//2) #1.0
```

eg-2:

```
print(4/2) #2.0
print(4//2) #2
print(4//2.0) #2.0
```

eg-3:

```
print(10+2) #12
print(10-2) #8
print(10*2) #20
print(10/2) #5.0
```

print(10//2) #5

print(10%2) #0

print(102) #100**

All these python's arithmetic operators are applicable on Numpy Arrays also.

1. Arithmetic operators for Numpy arrays with scalar:

----

scalar means constant numeric value.

All arithmetic operators are applicable for Numpy arrays with scalar.

eg-1: 1D array:

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

Similarly we can perform operations on any other dimension arrays.

eg-2: 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)
```

*****Note:**

In Python, divide by zero is not possible. If we are trying to perform, then we will get ZeroDivisionError.

```
print(0/0) # ZeroDivisionError: division by zero
print(10/0) #ZeroDivisionError: division by zero
```

But in Numpy we won't get ZeroDivisionError. If the result is undefined (0/0) then it is treated as nan (not a number). If the result is infinity (10/0) then it is treated as inf.

eg:

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

eg-2:

```
>>> a/np.inf
array([0., 0., 0., 0., 0., 0.])
```

2. Arithmetic operators for Arrays with Arrays (Arrays with Arrays):

---

eg-1:

```
>>> a = np.array([100,200,300,400])
>>> a
array([100, 200, 300, 400])
>>> a+a
array([200, 400, 600, 800])
>>> a-a
array([0, 0, 0, 0])
>>> a*a
array([ 10000,  40000,  90000, 160000])
>>> a/a
array([1., 1., 1., 1.])
>>> a//a
array([1, 1, 1, 1], dtype=int32)
>>> a%a
array([0, 0, 0, 0], dtype=int32)
>>> a**a
array([0, 0, 0, 0], dtype=int32)
```

eg-2:

```
>>> a = np.array([100,200,300,400])
>>> b = np.array([10,20,30,40])
>>> a
array([100, 200, 300, 400])
>>> b
array([10, 20, 30, 40])
>>> a+b
```

```
array([110, 220, 330, 440])
>>> a-b
array([ 90, 180, 270, 360])
>>> a*b
array([ 1000,  4000,  9000, 16000])
>>> a/b
array([10., 10., 10., 10.])
>>> a//b
array([10, 10, 10, 10], dtype=int32)
>>> a%b
array([0, 0, 0, 0], dtype=int32)
>>> a**b
array([1661992960,          0,          0,          0], dtype=int32)
```

eg-3: 2-D array

```
>>> a = np.array([[1,2,3],[4,5,6]])
>>> a
array([[1, 2, 3],
       [4, 5, 6]])
>>> a+a
array([[ 2,  4,  6],
       [ 8, 10, 12]])
>>> a-a
array([[0, 0, 0],
       [0, 0, 0]])
>>> a*a
array([[ 1,  4,  9],
       [16, 25, 36]])
>>> a/a
```

```
array([[1., 1., 1.],
       [1., 1., 1.]])
>>> a//a
array([[1, 1, 1],
       [1, 1, 1]], dtype=int32)
>>> a%a
array([[0, 0, 0],
       [0, 0, 0]], dtype=int32)
>>> a**a
array([[   1,    4,   27],
       [ 256,  3125, 46656]], dtype=int32)
```

Note: To perform arithmetic operators between numpy arrays, compulsory both arrays should have same dimension, same shape and same size, otherwise we will get error.

eg:

```
>>> a = np.array([10,20,30])
>>> b = np.array([40,50,60,70])
>>> a+b
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: operands could not be broadcast together with shapes (3,) (4,)
```

To handle this type of requirements we should go for broadcasting.

Note:

For every arithmetic operator numpy library defines equivalent functions and we can use these functions also.

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)**

eg-1:

```
>>> a = np.array([10,20,30,40])
>>> b = np.array([1,2,3,4])
>>> np.add(a,b)
array([11, 22, 33, 44])
>>> np.subtract(a,b)
array([ 9, 18, 27, 36])
>>> np.multiply(a,b)
array([ 10,  40,  90, 160])
>>> np.divide(a,b)
array([10., 10., 10., 10.])
>>> np.mod(a,b)
array([0, 0, 0, 0], dtype=int32)
>>> np.power(a,b)
array([    10,    400,   27000, 2560000], dtype=int32)
>>> np.floor_divide(a,b)
array([10, 10, 10, 10], dtype=int32)
>>> np.add(a,1)
array([11, 21, 31, 41])
```

Note: To use these functions also, both arrays should be of same dimension, same shape and same size.

Note: The functions which opearates element by element on whole array, are called universal functions(ufunc). All the above functions are universal functions.

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.

Note: If both arrays have same dimension, same shape and same size then broadcasting is not required. Different dimensions or different shapes or different sizes then only broadcasting is required.

Rules of Broadcasting:

Rule-1. If the two arrays are of different dimensions, numpy will make equal dimensions. Padded 1's in the shape of fewer dimension array on the left side.

eg:

a: (4,3)

b: (3,)

a is 2-D array where as b is 1-D array. Broadcasting will make these two arrays as 2-D arrays.

a: (4,3)

b: (1,3)

Now both arrays should be same dimensions(2-Dimensions).

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/increased to match other size of the same dimension.

Before:

a: (4,3)

b: (1,3)

After:

a: (4,3)

b: (4,3)

Now both arrays are of same dimension and same shape, then airthmetic operation will be performed normally element wise.

Note:

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

eg:

a: (4,3)

b: (2,3)

Rule 2 fails and hence we will get ValueError and broadcating won't be happend.

2. The elements of fewer size/deminsion array will be reused.

3. The result is always of higher dimension of input arrays.

inputs: 3-D, 1-D

output: 3-D

Note:

Rule-1: to make dimensions same

Rule-2: To make same sizes in every dimension

eg-1:

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

a: (4,)

b: (3,)

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

```
>>> a = np.array([10,20,30,40])
>>> b = np.array([1,2,3])
>>> a.shape
(4,)
>>> b.shape
(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, because both arrays are of same dimension

Rule-2:

Before: a' shape (3,) and b's shape (1,)

After: a' shape (3,) and b's shape (3,)

Now

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

b = np.array([40,40,40])

a+b will be performed

```
>>> a = np.array([10,20,30])
>>> b = np.array([40])
>>> a.shape
(3,)
>>> b.shape
(1,)
>>> a+b
array([50, 60, 70])
```

eg-3:

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

b = np.array([10,20])

Dimensions are not same and shapes are not same

Rule-1:

Before: a's shape:(3,2) and b's shape:(2,)

After: a's shape:(3,2) and b's shape:(1,2)

Rule-2:

Before: a's shape:(3,2) and b's shape:(1,2)

After: a's shape:(3,2) and b's shape:(3,2)

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

>>> 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+b
array([[20, 40],
       [40, 60],
       [60, 80]])
```

eg-4:

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

a+b

Rule-1:

a:(3,1)

b:(1,3)

Rule-2:

a:(3,3)

b:(3,3)

Note:

1. In broadcasting if rows are required to increase, then repeat existing row.

1. In broadcasting if columns are required to increase, then repeat existing column.

Q: Is it possible to perform arithmetic operation between 2 arrays of shapes: (1,2,3) and (4,5) ?

Even broadcasting also won't work in this case. Hence we will get error.

Array Manipulation Functions

----

1. The Complete story of reshape() function:

We can use reshape() function to change array shape without changing data.

Syntax:

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

Gives a new shape to an array without changing its data.

newshape : int or tuple of ints

order : {'C', 'F', 'A'}

Default value for the order: 'C'

'C' --->C language style which means row major order.

'F' --->Fortran language style which means column major order.

eg-1: To convert 1-D array to 2-D array:

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

Here we didn't specify order and hence default value 'C' is considered, which is meant for row major order.

eg-1:

```
>>> c = np.reshape(a,(5,2),order='F')
>>> c.shape
(5, 2)
>>> a
```

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

While reshaping, column wise order considered which is F-style.

Conclusion-1:

While using reshape() function, make sure the sizes should be matched otherwise we will get error.

```
>>> a
array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10])
>>> b = np.reshape(a,(5,3))
ValueError: cannot reshape array of size 10 into shape (5,3)
```

Conclusion-2:

----

reshape() won't provide a separate new array object, it will provide just view of the existing array because there is no change in data.

Hence if we perform any change in the reshaped array, it will be reflected in the original array and vice-versa also.

```
>>> a = np.arange(1,16)
>>> a
```

```
array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15])
>>> b = np.reshape(a,(5,3))
>>> b
array([[ 1,  2,  3],
       [ 4,  5,  6],
       [ 7,  8,  9],
       [10, 11, 12],
       [13, 14, 15]])
>>> b[0][0]=7777
>>> b
array([[7777,    2,    3],
       [   4,    5,    6],
       [   7,    8,    9],
       [  10,   11,   12],
       [  13,   14,   15]])
>>> a
array([7777,    2,    3,    4,    5,    6,    7,    8,    9,   10,   11,
         12,   13,   14,   15])
>>> a[1]=9999
>>> a
array([7777, 9999,    3,    4,    5,    6,    7,    8,    9,   10,   11,
         12,   13,   14,   15])
>>> b
array([[7777, 9999,    3],
       [   4,    5,    6],
       [   7,    8,    9],
       [  10,   11,   12],
       [  13,   14,   15]])
```

Note: ndarray class also contains reshape() method and hence we can call this method on any ndarray object.

numpy--->module

ndarray ---->class present in numpy module

reshape()-->method present in ndarray class

reshape()--->function present in numpy module.

numpy.reshape()--->numpy library function

b = np.reshape(a,shape,order)

ndarray.reshape()---->ndarray object method

b = a.reshape(shape,order)

eg-1:

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

eg-2:

```
>>> c = a.reshape((2,3,4),order='F')
```

```
>>> c
array([[[ 1,  7, 13, 19],
        [ 3,  9, 15, 21],
        [ 5, 11, 17, 23]],

       [[ 2,  8, 14, 20],
        [ 4, 10, 16, 22],
        [ 6, 12, 18, 24]]])
```

-1 story in unknown dimension:

If we don't know the size of any dimension, we can use -1 so that numpy itself will evaluate that size.

From Documentation:

One shape dimension can be -1. In this case, the value is inferred from the length of the array and remaining dimensions automatically by numpy itself.

```
b = a.reshape((5,-1)) #valid
b = a.reshape((-1,5)) #valid
b = a.reshape((-1,-1)) #invalid
```

Note: Instead of -1, we can use any negative int value.

eg-1:

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

```
>>> b.shape
(5, 2)
```

eg-2:

```
>>> b = a.reshape((-1,5))
>>> b.shape
(2, 5)
```

eg-3:

```
>>> b = a.reshape((-1,3))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: cannot reshape array of size 10 into shape (3)
```

eg-4:

```
>>> b = a.reshape((-1,-1))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: can only specify one unknown dimension
```

eg-5:

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

(3, 2, 2)

Conclusions of reshape():

----

1. To reshape array without changing data.

2. The sizes must be matched.

3. We can use as numpy library function or ndarray class method.

4. It won't create a new array object, just we will get view.

5. We can use -1 in the case of unknown dimension, but only once.

The complete story of resize() function:

Syntax:

numpy.resize(a, new_shape)

Return a new array with the specified shape.

If 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'.

eg-1:

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

eg-2:

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

Conclusions:

1. A separate new array object will be created.

2. There may be a chance of changing data.

3. The sizes are need not be same.

4. There is no chance to use -1 in the case of unknown dimension.

```
>>> c = np.resize(a,(2,-1))
 ValueError: all elements of 'new_shape' must be non-negative
```

*****Note:**

If we are using ndarray class resize() method, inline modification will be happend. ie existing array only modifed. If new_shape requires more elements then extra elements filled with zeros.

eg-1:

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

Differences between numpy.resize() and ndarray.resize()

----

table: numpy.resize() | ndarray.resize()

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(inline modification).

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.

Differences between reshape() and resize() :

Table: reshape() | resize()

----

1. It is to change shape of array, but not size.

1. It is to change size of the array, automatically shape and data may be changed.

2. 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.

2. 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.

3. The reshape will be happend without changing original data.

3. There may be a chance of data change in resize()

4. The sizes must be matched.

4. The sizes need not be matched.

5. In unknown dimension we can use -1.

5. -1, such type of story not applicable.

3. Array Manipulation by using flatten():

We can use flatten() method to flatten(convert) any n-Dimensional array to 1-Dimensional array.

Syntax:

ndarray.flatten(order='C')

It will create a new 1-Dimensional array with elements of given n-Dimensional array. ie Return a copy of the array collapsed into one dimension.

eg-1: 2-D to 1-D

```
>>> a = np.arange(1,11).reshape(5,2)
>>> a
array([[ 1,  2],
       [ 3,  4],
       [ 5,  6],
       [ 7,  8],
       [ 9, 10]])
>>> b = a.flatten()
>>> b
array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10])
>>> c = a.flatten(order='F')
```

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

eg-2: 3-D to 1-D

```
>>> a = np.arange(1,19).reshape(3,3,2)
>>> 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])
>>> c = a.flatten(order='F')
>>> c
array([ 1,  7, 13,  3,  9, 15,  5, 11, 17,  2,  8, 14,  4, 10, 16,  6, 12,
       18])
```

Note:

1. flatten will always create a new array object. Hence on the flatten copy, if we perform any changes then those changes won't be reflected to original copy.

eg:

```
>>> a = np.arange(1,7).reshape(3,2)
>>> a
array([[1, 2],
       [3, 4],
       [5, 6]])
>>> b = a.flatten()
>>> b
array([1, 2, 3, 4, 5, 6])
>>> b[0]=777
>>> b
array([777,   2,   3,   4,   5,   6])
>>> a
array([[1, 2],
       [3, 4],
       [5, 6]])
```

2. flatten() will always returns 1-D array only.

==========

flat variable:

It is a 1-D iterator over the array.

This is a 'numpy.flatiter' instance.

eg:

```
>>> a = np.arange(1,7).reshape(3,2)
>>> a
array([[1, 2],
       [3, 4],
       [5, 6]])
>>> a.flat
```

```
<numpy.flatiter object at 0x000001E57F57E0F0>
>>> a.flat[4]
5
>>> for x in a.flat: print(x)
...
1
2
3
4
5
6
```

3. The complete story 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.

Syntax:

```
numpy.ravel(a, order='C')
```

eg-1:

```
>>> a = np.arange(1,7).reshape(3,2)
>>> b = np.ravel(a)
>>> a
array([[1, 2],
       [3, 4],
       [5, 6]])
>>> b
array([1, 2, 3, 4, 5, 6])
>>> b[0]=7777
```

```
>>> b
array([7777,   2,   3,   4,   5,   6])
>>> a
array([[7777,   2],
       [  3,   4],
       [  5,   6]])
```

Note: For numpy library ravel() function, ndarray class contains equivalent ravel() method. We can use anything, functionality is exactly same.

eg:

```
>>> a = np.arange(1,19).reshape(2,3,3)
>>> a
array([[[ 1,  2,  3],
        [ 4,  5,  6],
        [ 7,  8,  9]],

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

Difference between flatten() and ravel():

Table: flatten() | ravel()

1. It can used to flatten n-D array to 1-D array and a separate 1-D array object will be created.

1. It can used to flatten n-D array to 1-D array but it won't create a new 1-D array object and we will get only view.

2. If we perform any changes in the flatten() copy, then those changes won't be reflected in the original copy.

2. If we perform any changes in the ravel() copy, then those changes will be reflected in the original copy.

3. flatten() method operates slower than ravel() as it is required to create a new array object.

3. ravel() method operates faster than flatten() as it is not required to create a new array object and it just returns a view.

4. flatten() is not numpy library level function and it is a method present in ndarray class.

ndarrayObj.flatten() --->valid

numpy.flatten(a) --->invalid

4. ravel() is both numpy library level function and ndarray class method.

ndarrayObj.ravel() --->valid

numpy.ravel(a) --->valid

The complete story of transpose() function:

In our general mathematics, how to get transpose of given matrix?

By interchanging rows and columns.

Similarly, to interchange dimensions of nd array, we should go for transpose() function.

Syntax:

----

numpy.transpose(a,axes=None)

Reverse or permute the axes of an array; returns the modified array(View but not copy)

For an array a with two axes, transpose(a) gives the matrix transpose.

If we are not providing axes argument value, then the dimensions simply reversed.

In transpose() operation, just dimensions will be interchanged, but not content. Hence it is not required to create new array and it returns view of the existing array.

eg:

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

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

(2,3,4):

2--->2 2-Dimensional arrays

3 --->In every 2-D array, 3 rows

4 --->In every 2-D array, 4 columns

24--->Total number of elements

If we transpose this 3-D array to (4,3,2):

4 ---> 4 2-D arrays

3 --->In every 2-D array, 3 rows

2 --->In every 2-D array, 2 columns

24--->Total number of elements

Q. What is the difference between reshape and transpose?

In reshape we can change the size of dimension, but total size should be same.

eg: input: (3,4)

output: (4,3),(2,6),(6,2),(1,12),(12,1),(2,2,3),(3,2,2)

But in transpose just we are interchanging the dimensions,but we won't change the size of any dimension.

eg: input: (3,4)

output: (4,3)

eg: input: (2,3,4)

output: (4,3,2),(2,4,3),(3,2,4),(3,4,2) but we cannot take(2,12),(3,8)

Note: transpose is special case of reshape.

eg-1: transpose of 2-D array:

```
>>> a = np.arange(1,7).reshape(3,2)
>>> a
array([[1, 2],
       [3, 4],
       [5, 6]])
>>> b= np.transpose(a)
>>> b
array([[1, 3, 5],
       [2, 4, 6]])
```

In 2-D array, because of transpose, rows will become columns and columns will become rows.

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

In transpose, we will get only view part.

```
>>> b[0][0]=7777
```

```
>>> b
array([[7777,   3,   5],
       [  2,   4,   6]])
>>> a
array([[7777,   2],
       [  3,   4],
       [  5,   6]])
```

eg-2: transpose of 3-D array:

```
>>> a = np.arange(1,25).reshape(2,3,4)
>>> 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]]])
```

2--->2-D arrays

3--->3 rows in every 2-D array

4--->4 columns in every 2-D array

If we perform transpose operation without axes argument, then dimensions will be simply reversed.

4---> 4 2-D arrays

3--->3 rows in every 2-D array

2---->2 columns in every 2-D array

```
>>> b = np.transpose(a)
>>> b
array([[[ 1, 13],
        [ 5, 17],
        [ 9, 21]],

       [[ 2, 14],
        [ 6, 18],
        [10, 22]],

       [[ 3, 15],
        [ 7, 19],
        [11, 23]],

       [[ 4, 16],
        [ 8, 20],
        [12, 24]]])
```

eg-3: transpose of 1-D array:

Transposing a 1-D array returns an unchanged view of the original array.

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

axes parameter:

If we are not using axes parameter, then dimensions simply reversed.

axes parameter describes in which order we have to take axes.

It is very helpful for 3-D and 4-D arrays.

eg-1: for 2-D array:

----

input: (3,4)

The size of axis-0 is 3.

The size of axis-1 is 4.

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

No change in the result.

```
>>> a = np.arange(1,13).reshape(3,4)
>>> a
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12]])
>>> b = np.transpose(a,axes=(0,1))
>>> b
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12]])
```

input: (3,4)

The size of axis-0 is 3.

The size of axis-1 is 4.

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

In the result array, axis-0 size is axis-1 size of input.

In the result array, axis-1 size is axis-0 size of input.

Output:(4,3)

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

eg-2: for 3-D array:

---

input: (2,3,4)

The size of axis-0 is:2

The size of axis-1 is:3

The size of axis-2 is:4

```
>>> a = np.arange(1,25).reshape(2,3,4)
>>> 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]]])

b = np.transpose(a,axes=(2,0,1))

The shape of b: (4,2,3)

4 --->2-D arrays

2---->2 rows in every 2-D array

3---->3 columns in every 2-D array

>>> b = np.transpose(a,axes=(2,0,1))

>>> b

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

[13, 17, 21]],

[[2, 6, 10],

[14, 18, 22]],

[[3, 7, 11],

[15, 19, 23]],

[[4, 8, 12],

[16, 20, 24]]])

input: (2,3,4)

The size of axis-0 is:2

The size of axis-1 is:3

The size of axis-2 is:4

b = np.transpose(a,axes=(1,2,0))

The shape of b:(3,4,2)

```
>>> b = np.transpose(a,axes=(1,2,0))
>>> b
array([[[ 1, 13],
        [ 2, 14],
        [ 3, 15],
        [ 4, 16]],

       [[ 5, 17],
        [ 6, 18],
        [ 7, 19],
        [ 8, 20]],

       [[ 9, 21],
        [10, 22],
        [11, 23],
        [12, 24]]])
```

***Note: If we repeat the same axis multiple times then we will get error.

```
>>> b = np.transpose(a,axes=(2,2,1))
ValueError: repeated axis in transpose
```

ndarray class transpose() method:

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

Syntax:

```
a.transpose(*axes)
```

axes : None, tuple of ints, or 'n' ints

eg:

```
>>> a = np.arange(1,25).reshape(2,3,4)
>>> b = a.transpose()
>>> b.shape
(4, 3, 2)
>>> b = a.transpose((2,0,1))
>>> b.shape
(4, 2, 3)
```

Note: To get transpose we can use

a.T ---->Simply reverse the dimensions

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

T facility applicable only for reverse.

Hence, The following 2 lines are equal:

1. **a.transpose()**
2. **a.T**

Note: 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.

The complete story of swapaxes():

By using transpose we can interchange 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)

Interchange two axes of an array.

Parameters

---

a : array_like

Input array.

axis1 : int

First axis.

axis2 : int

Second axis.

eg-1:

```
>>> a = np.arange(1,7).reshape(3,2)
>>> a
array([[1, 2],
       [3, 4],
       [5, 6]])
>>> b = np.swapaxes(a,0,1)
```

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

Q. What is the difference between the two lines for 2-D array, a:

```
b = np.swapaxes(a,0,1)
b = np.swapaxes(a,1,0)
```

There is no difference between these lines just interchange rows and columns.

eg-2:

input: 3-D array of (2,3,4)

```
b = np.swapaxes(a,0,2)
b.shape --->(4,3,2)

b = np.swapaxes(a,1,0)
b.shape --->(3,2,4)
```

```
>>> a = np.arange(1,25).reshape(2,3,4)
>>> b = np.swapaxes(a,0,2)
>>> b.shape
(4, 3, 2)
>>> b = np.swapaxes(a,0,1)
>>> b.shape
```

(3, 2, 4)

ndarray.swapaxes():

----

ndarray class also contains swapaxes() method which is exactly same as numpy library swapaxes() function.

a.swapaxes(axis1, axis2)

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

numpy.swapaxes : equivalent function

eg:

>>> a = np.arange(1,25).reshape(2,3,4)

>>> b = a.swapaxes(1,2)

>>> b.shape

(2, 4, 3)

Difference between transpose() and swapaxes():

By using tarnspose() we can interchange any number of dimensions.

But by using swapaxes() we can interchange only two dimensions.

Joining of multiple arrays into a single array:

It is similar to join queries in Oracle.

We can join/concatenate multiple ndarrays into a single array by using the following functions.

1. concatenate()

2. stack()

3. vstack()

4. hstack()

5. dstack()

1. joining by using concatenate() function:

Syntax:

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

Join a sequence of arrays along an existing axis.

(a1, a2, ...)--->input arrays for concatenation

axis--->based on which axis, we have to perform concatenation. If axis = None, then arrays will be flatten (converts to 1-D array) and then perform concatenation.

out-->destination array, where we have to store concatenation result.

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 axis should be same.

3. The shapes of resultant array and out array must be same.

eg-1: Concatenation of two 1-D arrays

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

eg-2: Concatenation of three 1-D arrays

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

eg-3: storing result by using out parameter:

```
>>> a = np.arange(4)
>>> b = np.arange(5)
>>> c = np.arange(3)
>>> d = np.zeros(12)
>>> np.concatenate((a,b,c),out=d)
array([0., 1., 2., 3., 0., 1., 2., 3., 4., 0., 1., 2.])
>>> d
array([0., 1., 2., 3., 0., 1., 2., 3., 4., 0., 1., 2.])
```

Note: The default data type for zeros() is float and hence the elements are of float type.

If we want int type:

```
>>> d = np.zeros(12,dtype='int32')
>>> np.concatenate((a,b,c),out=d)
array([0, 1, 2, 3, 0, 1, 2, 3, 4, 0, 1, 2])
>>> d
array([0, 1, 2, 3, 0, 1, 2, 3, 4, 0, 1, 2])
```

Note: Compulsory the out parameter shape and resultant array shape must be matched, otherwise we will get error.

```
>>> d = np.zeros(10)
```

```
>>> np.concatenate((a,b,c),out=d)
ValueError: Output array is the wrong shape
```

Q. Is it possible to join a 1-D array and a 2-D array?

```
>>> a = np.arange(5)
>>> b = np.arange(12).reshape(3,4)
>>> np.concatenate(a,b)
TypeError: only integer scalar arrays can be converted to a scalar index
```

Using dtype parameter:

```
>>> a = np.arange(6)
>>> b = np.arange(3)
>>> np.concatenate((a,b),dtype=str)
array(['0', '1', '2', '3', '4', '5', '0', '1', '2'], dtype='<U11')
```

eg-4: joining of two 2-D arrays:

```
>>> a = np.array([[10,20],[30,40],[50,60]])
>>> a.shape
(3, 2)
>>> b = np.array([[70,80],[90,100]])
>>> b.shape
(2, 2)
>>> c = np.concatenate((a,b))
>>> c
array([[ 10,  20],
       [ 30,  40],
       [ 50,  60],
```

```
     [ 70,  80],
     [ 90, 100]])
>>> np.concatenate((a,b),axis=0)
array([[ 10,  20],
     [ 30,  40],
     [ 50,  60],
     [ 70,  80],
     [ 90, 100]])
>>> np.concatenate((a,b),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 3 and the array at index 1 has size 2
```

Note:

np.concatenate((a,b))--->valid

np.concatenate((a,b),axis=0)--->valid

np.concatenate((a,b),axis=1)--->invalid

np.concatenate((a.T,b),axis=1)--->valid

a.T --->(2,3)

b----> (2,2)

```
>>> np.concatenate((a.T,b),axis=1)
array([[ 10,  30,  50,  70,  80],
     [ 20,  40,  60,  90, 100]])
```

eg-4:

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

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

Note:

If axis = None, then arrays will be flatten (converts to 1-D array) and then perform concatenation.

```
>>> a
array([[10, 20],
     [30, 40],
     [50, 60]])
>>> b
array([[ 70,  80],
     [ 90, 100]])
>>> np.concatenate((a,b),axis=None)
array([ 10,  20,  30,  40,  50,  60,  70,  80,  90, 100])
```

eg: for 3-D array:

----

a=(2,3,2)

b=(2,3,3)

We can concatenate but we should use axis: 2.

ouput: (2,3,5)

```
>>> 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]]])
>>> c = np.concatenate((a,b),axis=2)
>>> c.shape
(2, 3, 5)
>>> c
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. Sir if we have 2 arrays....2,3,3 and 1,3,3 then how will the arrays concatenate on axis 0?(diagramatically)

Yes possible. The resultant array shape: (3,3,3)

Q. Is it possible to concatenate arrays with shapes (3,2,3) and (2,1,3)?

Not possible to concatenate on any axis.

But axis=None is possible. In this case both arrays will be flatten to 1-D and then concatenation will be happend.

Joining Numpy arrays by using stack() function:

Syntax:

----

stack(arrays, axis=0, out=None)

Join a sequence of arrays along a new axis.

The 'axis' parameter specifies the index of the new axis in the dimensions of the result. For example, if axis=0 it will be the first dimension and if axis=-1 it will be the last dimension.

The input arrays must have same shape.

The resultant stacked array has one more dimension than the input arrays.

eg-1: For stacking of 1-D arrays:

To use stack() method, make sure all input arrays must have same shape, otherwise we will get error.

```
>>> a = np.array([10,20,30])
>>> b = np.array([40,50,60,70])
>>> np.stack((a,b))
ValueError: all input arrays must have the same shape
```

axis-0: row wise:

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

input arrays are 1-D arrays of same shape.

The output array(stacked array) will be of 2-D

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

In 2-D array, stack row-wise all input arrays.

axis-0 means stack all input arrays row-wise.

```
>>> np.stack((a,b),axis=0)
array([[10, 20, 30],
       [40, 50, 60]])
>>> np.stack((a,b))
array([[10, 20, 30],
       [40, 50, 60]])
```

axis-1: column wise:

---

axis-1 means stack elements of input array column wise

Read row wise from input arrays and arrange column wise in result array.

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

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

eg-1: For stacking of 2-D arrays:

If we stack 2-D arrays the result 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 2-D array

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

axis-0 means 2-D arrays one by one:

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

       [[ 7,  8,  9],
```

[10, 11, 12]]])

axis-1 means row wise in each 2-D array:

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

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

[7, 8, 9]],

[[4, 5, 6],

[10, 11, 12]]])

axis-2 means column wise in each 2-D array:

----

take each row in input array and make as column

>>> np.stack((a,b),axis=2)

array([[[1, 7],

[2, 8],

[3, 9]],

[[4, 10],

[5, 11],

[6, 12]]])

eg-3: for 2-D arrays:

If the input is 2-D array, then output will become 3-D arrays.

>>> a = np.arange(1,10).reshape(3,3)

>>> b = np.arange(10,19).reshape(3,3)

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

axis-0 means 2-D arrays one by one:
-----------------------------------
>>> np.stack((a,b))
array([[[ 1,  2,  3],
        [ 4,  5,  6],
        [ 7,  8,  9]],

       [[10, 11, 12],
        [13, 14, 15],
        [16, 17, 18]]])
axis-1 means row wise in each 2-D array:
----------------------------------------
>>> np.stack((a,b),axis=1)
array([[[ 1,  2,  3],
        [10, 11, 12]],

       [[ 4,  5,  6],
        [13, 14, 15]],

       [[ 7,  8,  9],
```

[16, 17, 18]]])

axis-2 means column wise in each 2-D array:

----

take each row in input array and make as column

>>> np.stack((a,b),axis=2)

array([[[1, 10],

[2, 11],

[3, 12]],

[[4, 13],

[5, 14],

[6, 15]],

[[7, 16],

[8, 17],

[9, 18]]])

eg-4: stacking of three 1-D arrays:

>>> a = np.arange(4)

>>> b = np.arange(4,8)

>>> c = np.arange(8,12)

>>> a

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

>>> b

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

>>> c

array([8, 9, 10, 11])

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

eg-5: stacking of three 2-D arrays:

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

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

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

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

       [[ 2,  6, 10],
        [ 3,  7, 11]]])
```

Joining of arrays by using vstack() function:

---

vstack --->vertical stack (axis=0) row wise

vstack(tuple of arrays)

Stack arrays in sequence vertically (row wise).

Rules:

1. The input arrays must have the same shape along all except first axis(axis-0)

2. 1-D arrays must have the same size.

3. The array formed by stacking the given arrays, will be at least 2-D.

4. vstack() operation is equivalent to concatenation along the first axis after 1-D arrays of shape (N,) have been reshaped to (1,N).

5. For 2-D or more dimension arrays, vstack() simply acts as concatenation wrt axis-0.

eg-1: For 1-D arrays:

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

a will be converted to shapes (1,4) and b will be converted to (1,4)

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

eg-2: for 1-D arrays of different sizes:

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

a will be converted to shapes (1,4) and b will be converted to (1,6)

```
>>> a = np.array([10,20,30,40])
>>> b = np.array([50,60,70,80,90,100])
>>> 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 4 and the array at index 1 has size 6

Note: To use vstack() function for 1-D arrays, compulsory input arrays must be of same size/shape.

eg-3: For 2-D arrays:

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

eg-4: For 2-D arrays:

```
>>> 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-5: For 3-D arrays:

Diagram

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

```
>>> 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]]])
>>> b
array([[[25, 26, 27, 28],
        [29, 30, 31, 32],
        [33, 34, 35, 36]],

       [[37, 38, 39, 40],
        [41, 42, 43, 44],
        [45, 46, 47, 48]]])
>>> 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:

----

hstack --->horizontal(column wise) axis-1

Syntax:

hstack(tuple of arrays)

Stack arrays in sequence horizontally (column wise).

Rules:

1. This is equivalent to concatenation along the second axis, except for 1-D arrays where it concatenates along the first axis.

2. All input arrays must be same dimension.

3. Except axis-1, all remining sizes must be equal.

eg-1: For 1-D arrays:

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

eg-3: for 2-D arrays:

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

Joining of arrays by using dstack() function:

---

dstack() --->depth/height stack --->concatenation along 3rd axis

Syntax:

dstack(tuple of input arrays)

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

Rules:

1. **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).**
2. **The arrays must have the same shape along all but the third axis.**
   1-D or 2-D arrays must have the same shape.
3. **The array formed by stacking the given arrays, will be at least 3-D.**

eg-1 :

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

eg-2:

```
>>> 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).

Numpy--->ndarrays

Pandas--->Series, DataFrame

Scipy

Matplotlib-->10 lines of code

seaborn--->4 lines of code

plotly-->2 lines of code

Splitting of arrays:

We can split an array into multiple subarrays as views.

1. split()

2. vsplit()

3. hsplit()

4. dsplit()

5. array_split()

Splitting by using split() function:

Syntax:

split(array, indices_or_sections, axis=0)

--->Split an array into multiple sub-arrays of equal size.

-->sections means the number of sub-arrays

--->it returns list of ndarray objects.

--->all sections must be of equal size, otherwise error.

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

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

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

>>> type(sub_arrays)

<class 'list'>

>>> sub_arrays[0]

array([1, 2, 3])

>>> sub_arrays[1]

array([4, 5, 6])

Note: If dividing array into equal number of specified sections is not possible, then we will get error.

>>> a

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

>>> np.split(a,4)

ValueError: array split does not result in an equal division

Dividing 9 elements into 4 equal parts is not possible and hence we got error.

eg-2: splitting of 2-D array:

In case of 2-D array, splitting will be happend based on axis-0 bydefault. ie row wise spliting(vertical spliting)

But we can specify axis-1, then splitting will be happend based columns (horizontal spliting)

>>> 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,3) #dividing a into 3 sections vertically(row-wise)
[ 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,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]])]
```

Note: Here we can use various possible sections: 2,3,6

```
>>> 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]])]
```

splitting based on axis-1:

```
--------------------------
>>> 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) # division happend horizontally(column wise)
[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]])]
```

Note: We can split any n-Dimensional arrays based on specified axis.

split based on indices:

We can also split based on indices. The sizes of sub-arrays are need not be equal.

eg-1:

```
>>> a = np.arange(10,101,10)
>>> a
array([ 10,  20,  30,  40,  50,  60,  70,  80,  90, 100])
>>> np.split(a,[3,7])
[array([10, 20, 30]), array([40, 50, 60, 70]), array([ 80,  90, 100])]
```

Diagram

eg-2:

```
>>> a = np.arange(10,101,10)
>>> a
array([ 10,  20,  30,  40,  50,  60,  70,  80,  90, 100])
>>> np.split(a,[2,5,7])
[array([10, 20]), array([30, 40, 50]), array([60, 70]), array([ 80,  90, 100])]
```

eg-2:

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

eg-3:

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

Diagram

eg-4:

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

eg-5:

```
>>> np.split(a,[2,4,4],axis=1)
[array([[ 1,  2],
       [ 7,  8],
       [13, 14]]), array([[ 3,  4],
       [ 9, 10],
       [15, 16]]), array([], shape=(3, 0), dtype=int32), array([[ 5,  6],
       [11, 12],
       [17, 18]])]
```

eg-6:

```
>>> np.split(a,[0,2,6],axis=1)
[array([], shape=(3, 0), dtype=int32), array([[ 1,  2],
```

[7, 8],

[13, 14]]), array([[3, 4, 5, 6],

[9, 10, 11, 12],

[15, 16, 17, 18]]), array([], shape=(3, 0), dtype=int32)]

splitting by vsplit()

If we want to split based on first axis(axis-0) then we should go for vsplit()

vsplit--->vertical split (row wise)

Syntax:

vsplit(array, indices_or_sections)

Split an array into multiple sub-arrays vertically (row-wise).

eg-1: For 1-D array:

It is not possible to split 1-D array vertically.

```
>>> a = np.arange(10)
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> np.vsplit(a,2)
ValueError: vsplit only works on arrays of 2 or more dimensions
```

eg-2: For-2D array:

```
>>> 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,3)
[array([[1, 2],
       [3, 4]]), array([[5, 6],
       [7, 8]]), array([[ 9, 10],
       [11, 12]])]
```

eg-3: vsplit() based on indices:

```
>>> 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,[1,4])
[array([[1, 2]]), array([[3, 4],
       [5, 6],
       [7, 8]]), array([[ 9, 10],
       [11, 12]])]
```

splitting by hsplit():

---

hsplit--->means horizontal split(column wise)

split will be happend based on 2nd axis (axis-1)

Syntax:

------

hsplit(array,indices_or_sections)

Split an array into multiple sub-arrays horizontally (column-wise).

eg-1: for 1-D array

```
>>> 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])]
```

eg-2: for 2-D array:

```
>>> 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])]

eg-3: Based on indices:

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

>>> a

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

>>> np.hsplit(a,[2,4,7])

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

splitting by dsplit():

----

dsplit --->means depth split

splitting based on 3rd axis(axis-2).

Syntax:

dsplit(array, indices_or_sections)

-->Split array into multiple sub-arrays along the 3rd axis (depth).

--->'dsplit' is equivalent to 'split' with 'axis=2', the array is always split along the third axis provided the array dimension is greater than or equal to 3.

eg:

>>> a = np.arange(16).reshape(2,2,4)

>>> a

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

[4, 5, 6, 7]],

[[8, 9, 10, 11],

[12, 13, 14, 15]]])

>>> np.dsplit(a,2)

[array([[[0, 1],

[4, 5]],

[[8, 9],

[12, 13]]]), array([[[2, 3],

[6, 7]],

[[10, 11],

[14, 15]]])]

Diagram

splitting by using array_split():

In the case of split() with sections, the array should be splitted into equal parts. If equal parts are not possible, then we will get error.

But in the case of array_split() we won't get any error.

Syntax:

array_split(array, indices_or_sections, axis=0)

Split an array into multiple sub-arrays.

The only difference between split() and array_split() 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.

eg-1:

10 elements --->3 sections

10%3 sub-arrays of size 10//3+1 and the rest of size 10//3

1 sub-array of size 4 and the rest of size 3

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

2 sub arrays: 4 size

1 sub array:3

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

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.

Sorting of ndarrays:

We can sort elements of nd array.

numpy module contains sort() function.

Syntax:

sort(a)

The default sorting algorithm is Quick sort. We can also specify Merge sort, heapsort etc.

eg-1: To sort elements of 1-D array

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

Note: For numbers default sorting is ascending order.

To sort descending order:

1st way:

----

a: 70,20,60,10,50,40,30

-a: -70,-20,-60,-10,-50,-40,-30

sort(-a): -70,-60,-50,-40,-30,-20,-10

-sort(-a): 70,60,50,40,30,20,10

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

2nd way:

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

eg-2: To sort elements in alphabetical order

```
>>> a = np.array(['cat','rat','bat','dog'])
>>> a
array(['cat', 'rat', 'bat', 'dog'], dtype='<U3')
>>> b = np.sort(a)
>>> b
array(['bat', 'cat', 'dog', 'rat'], dtype='<U3')
>>> np.sort(a)[::-1]
array(['rat', 'dog', 'cat', 'bat'], dtype='<U3')
```

Sorting of 2-D array:

Every 1-D array will be sorted.

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

Searching elements of ndarray:

We can search elements of ndarray by using where() function.

Syntax:

where(condition, [x, y])

Return elements chosen from 'x' or 'y' depending on 'condition'.

out : ndarray

An array with elements from x where condition is True, and elements from y elsewhere.

If we are not providing x and y, then it returns ndarray of indexes where condition satisfied.

eg-1: Find indexes where the value is 7 from 1-D array

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

>>> b

(array([2, 4], dtype=int64),)

ie 7 is available at indices 2 and 4.

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

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

>>> b = np.where(a%2 != 0)

>>> b

(array([0, 1, 2, 4, 5, 9], dtype=int64),)

eg-3: Find indexes where even numbers present in the given 1-D array?

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

>>> b = np.where(a%2 == 0)

>>> b

(array([3, 6, 7, 8], dtype=int64),)

To get even elements condition based selection:

>>> evens=a[a%2 == 0]

>>> evens

array([6, 4, 6, 10])

Note:

where(condition,[x,y])

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

eg-3: Replace every even number with 8888 and every odd number with 7777

```
>>> a = np.array([3,5,7,6,7,9,4,6,10,15])
>>> b = np.where( a%2 == 0, 8888, 7777)
>>> b
array([7777, 7777, 7777, 8888, 7777, 7777, 8888, 8888, 8888, 7777])
```

eg-4: Find indexes where odd numbers present in the given 1-D array and replace with element 9999.

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

searchsorted() function:

To perform search operation, internall this function using Binary search algorithm.

Syntax:

searchsorted(a, v, side='left', sorter=None)

Find indices where elements should be inserted to maintain order.

To use this function, array should be sorted already otherwise we will get abnormal results.

eg-1:

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

1

>>> np.searchsorted(a,13)

3

eg-2:

By default it will always search from the left hand side to identify insertion point. If we want to search from the right hand side we should use side='right'.

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

>>> b = np.sort(a)

>>> b

array([3, 4, 5, 6, 6, 7, 7, 9, 10, 15])

>>> np.searchsorted(b,6)

3

>>> np.searchsorted(b,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?

We can insert elements into ndarrys by using the following functions.

1. insert()

2. append()

1. insert():

----

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

Insert values along the given axis before the given indices.

obj-->Object that defines the index or indices before which 'values' is inserted.

values--->Values to insert into array.

axis ---->Axis along which to insert 'values'.

Inserting elements into 1-D array:

----

eg-1: To insert 7777 before index 2

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

>>> 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 indexes 2 and 5.

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

eg-2: To insert 7777 before 2 and 8888 before 5.

```
>>> b = np.insert(a,[2,5],[7777,8888])
>>> b
array([  0,    1, 7777,    2,    3,    4, 8888,    5,    6,    7,    8,  9])
```

Q. What is the result of the following?

>>> 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,)

>>> np.insert(a,25,7777)

IndexError: index 25 is out of bounds for axis 0 with size 10

*****Note: Array should contains only homogeneous elements. By using insert() function, if we are trying to insert any other type element, then that element will be converted to array type automatically before insertion. If the conversion is not possible then we will get error.**

eg-1:

>>> a = np.arange(10)

>>> a

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

>>> np.insert(a,2,10.5)

array([0, 1, 10, 2, 3, 4, 5, 6, 7, 8, 9])

Here array contains int values, but we are trying to insert float value, which will be converted to int type automatically.

eg-2:

>>> np.insert(a,2,True) #True will be converted to 1

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

```
>>> np.insert(a,2,False) #False will be converted to 0
array([0, 1, 0, 2, 3, 4, 5, 6, 7, 8, 9])
```

eg-3:

```
>>> np.insert(a,2,10+20j)
TypeError: can't convert complex to int
```

eg-4:

```
>>> np.insert(a,2,'durga')
ValueError: invalid literal for int() with base 10: 'durga'
```

Inserting elements into 2-D arrays:

If we are trying to insert elements into multi dimensional arrays, compulsory we have to provide axis. If we are not providing axis value, then default value None will be considered. In this case, array will be flatten to 1-D array and then insertion will be happend.

eg-1:

```
>>> a = np.array([[1,2],[3,4]])
>>> a
array([[1, 2],
       [3, 4]])
>>> np.insert(a,1,10)
array([ 1, 10,  2,  3,  4])
```

eg-2:

```
>>> np.insert(a,1,10,axis=0)
array([[ 1,  2],
       [10, 10],
       [ 3,  4]])
```

eg-3:

```
>>> np.insert(a,1,10,axis=1)
array([[ 1, 10,  2],
       [ 3, 10,  4]])
```

eg-4:

```
>>> np.insert(a,1,[10,20],axis=0)
array([[ 1,  2],
       [10, 20],
       [ 3,  4]])
```

eg-5:

```
>>> np.insert(a,1,[10,20],axis=1)
array([[ 1, 10,  2],
       [ 3, 20,  4]])
```

eg-6:

```
>>> np.insert(a,1,[10,20,30],axis=0)
ValueError: could not broadcast input array from shape (1,3) into shape (1,2)
```

eg-7:

```
>>> np.insert(a,0,[10,20],axis=0)
array([[10, 20],
       [ 1,  2],
       [ 3,  4]])
```

Note: We can take negative values for index and axis.

```
>>> np.insert(a,0,[10,20],axis=-1)
array([[10,  1,  2],
       [20,  3,  4]])
```