DurgaSir_DS_Numpy_Complete

Chapter-1
Basic Introduction to the Numpy

Numpy Introduction

- **Numpy** ==> Numerical Python Library
- **Scipy** ==> Scientific Python Library

Numpy is the entry point for Data Science

Need of Numpy

Python performs basic mathematical operations.

But in Data Science, Machine Learning, Deep Learning and Artificail Intelligence require complex mathematical operations like

1. Creation of arrays
2. Perform several operations on those arrays
3. Integral calculus operations
4. Differential equaltions
5. Statistics related operations

These type of opertions are not supported by normal Python.
Numpy can perform all these operations.

Importance of Numpy

- **ndarray** (n- dimensional array) is the basic data structure in numpy.
- Numpy is the backboone of remaining libraries like pandas, matplotlib,sklearn(sci-kit learn) etc.
- Numpy has Vectorization features

Numpy

History Numpy

- Numpy stands for **Numerical Python**
- It is the fundamental python library to perfrom 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 mutiple contributors
- Numpy is freeware and open source library
- Numpy library is written in C and Python
- C language ==> performance is very high
- Most of Numpy is written in C, So performance-wise Numpy is the best
- Because of high speed, numpy is best choice for ML algorithms than traditionsl python's in-built data structures like List.

Features of Numpy

1. Superfast beacuse most of numpy is written in C language
2. Numpy Array(nd array) basic data structure in Numpy.
3. It is the backbone for remaining libraries like pandas, scikit-learn etc
4. Numpy has vectorization feature which imporves performance while iterating elements

ndarray in Numpy

- In numpy data is stored in the form of array.
- In numpy we can hold data by using Array Data Structure.
- The Arrays which are created by using numpy are called nd arrays
- nd array ==> N-Dimensional Array or Numpy Array
- This nd array is most commonly used in Data Science libraries like pandasm scikit learn etc.

Application Areas of Numpy

- To perform linear algebra functions
- To perform linear regression
- To perfom logistic regression
- Deep Neural Networks
- K-means clustering
- Control Systmes
- Operational Research

What is an Array

- An indexed collection of homogenous data elements is nothing but array.
- It is the most commonly used concept in programming lanaguage like C/C++, java etc.
- By default 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:

- By using array module
- By using numpy module

installing Numpy module

To install latest version

- **pip install numpy**

to install particular version of numpy

- **pip install numpy==1.20.0**

Note:

When we install Anaconda python. Then there is no need to install numpy explicitly. numpy module is implicitly installed along with Anaconda pyton

In [1]:

```
# Check the version of Numpy installed
import numpy as np
print(np.__version__)
```

```
1.20.1
```

Python List Vs Numpy Array

Similarities:

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

Differences:

1. list is python's inbuilt type. we have to install and import numpy explicitly.
2. List can contain heterogeneous elements. But array contains only homogeneous elements.
3. On list, we cannot perform vector operations. But on ndarray we can perform vector operations.
4. Arrays consume less memory than list.
5. Arrays are superfast when compared with list.
6. Numpy arrays are more convenient to use while performing complex mathematical operations

In [2]:

```
# 3. On list, we cannot perform vector operations.
# But on ndarray we can perform vector operations.
import numpy as np
l = [10,20,30,40]
a = np.array([10,20,30,40])
l+2 # not possible
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-2-5111be676974> in <module>
      4 l = [10,20,30,40]
      5 a = np.array([10,20,30,40])
----> 6 l+2 # not possible

TypeError: can only concatenate list (not "int") to list
```

In [3]:

```
# On ndarrays we can perform vector operations
import numpy as np
a = np.array([10,20,30,40])
a+2 # array([12,22,32,42])
```

Out[3]:

```
array([12, 22, 32, 42])
```

In [4]:

```
# 4. Arrays consume less memory than list.
import numpy as np
import sys
l = [10,20,30,40,50,60,70,80,90,100,10,20,30,40,50,60,70,80,90,100]
a = np.array([10,20,30,40,50,60,70,80,90,100,10,20,30,40,50,60,70,80,90,100])
print('The Size of list l => ',sys.getsizeof(l))
print('The Size of ndarray a => ',sys.getsizeof(a))
```

```
The Size of list l =>  216
The Size of ndarray a =>  184
```

In [6]:

```
# 5. Arrays are superfast when compared with list.
import numpy as np
from datetime import datetime

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

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

#traditional python code
def dot_product(a,b):
   result = 0
   for i,j in zip(a,b):
      result = result + i*j
   return result

before = datetime.now()
```

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

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

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

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

```
The Time taken by traditonal python: 0:00:03.665659
The Time taken by Numpy Library: 0:00:03.522690
```

```
In [7]:

import numpy as np
import time
import sys

SIZE =1000000

l1 = range(SIZE)
l2 = range(SIZE)

a1 = np.arange(SIZE)
a2 = np.arange(SIZE)

#python list
start  = time.time()
result = [ (x+y) for x,y in zip(l1,l2)]
# adds first element in l1 with first element of l2 and so on
print("Pyton list took : ",(time.time() - start)*1000)
```

```
#numpy array
start= time.time()
result = a1 + a2
print("Numpy array took : ",(time.time() - start)*1000)
```

```
Pyton list took :  300.19593238830566
Numpy array took :  29.91938591003418
```

Chapter-2
Creation of Numpy Arrays

Array (ndarray) creation using array module

In [8]:

```
import array
a = array.array('i',[10,20,30])  # i represents int type array
print(f"Type of a ==> {type(a)}") # array.array
print(f"a value ==> {a}")
print("Elements one by one")
for x in range(len(a)):
  print(a[x])
```

```
Type of a ==> <class 'array.array'>
a value ==> 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

Array (ndarray) creation using Numpy

1. array()
2. arange()
3. linspace()
4. zeros()
5. ones()
6. full()
7. eye()
8. identity()
9. diag()
10. empty()

11. np.random module
    - randint()
    - rand()
    - uniform()
    - randn()
    - normal()
    - shuffle()

Note:

- **np.random** module is developed based on array.
- So performance of this module is more when compared with normal **python random module**

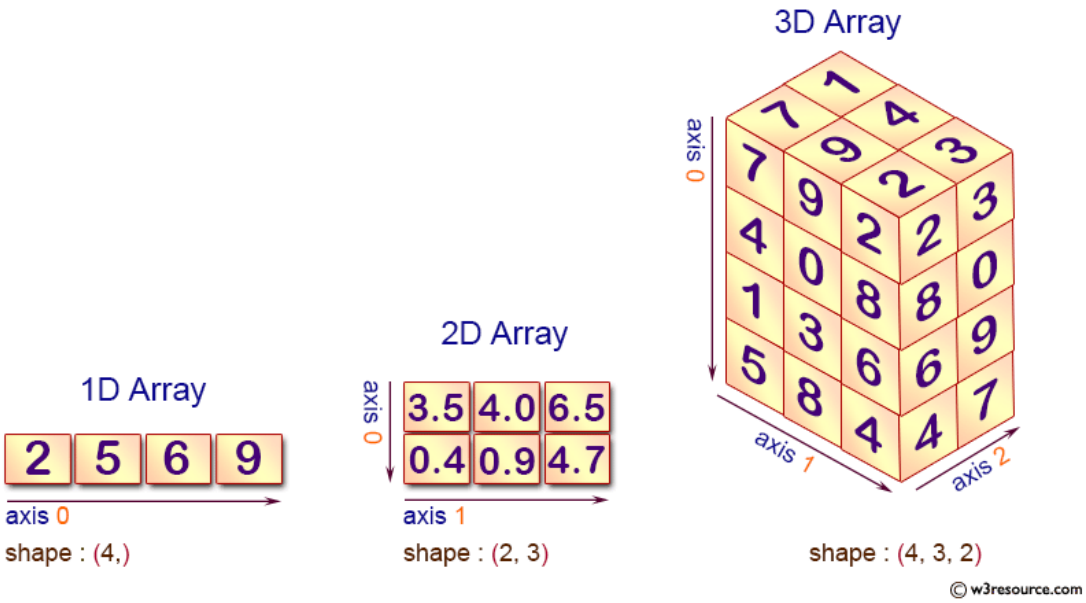
array()

In [9]:

```
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.

    Parameters
    ----------
    object : array_like
        An array, any object exposing the array interface, an object whose
        __array__ method returns an array, or any (nested) sequence.
    dtype : data-type, optional
        The desired data-type for the array.  If not given, then the type
        Will be determined as the minimum type required to hold the objects
        in the  sequence.

    subok : bool, optional
        If True, then sub-classes will be passed-through, otherwise
        the returned array will be forced to be a base-class array (default).
```

```
ndmin : int, optional
    Specifies the minimum number of dimensions that the resulting
    array should have.  Ones will be pre-pended to the shape as
    needed to meet this requirement.
```



1-D array(Vector) creation using list or tuple

In [10]:

```
# 1-D array creation from list or tuple
l = [10,20,30]
print(f'Type of l ==> {type(l)}')
a = np.array(l)
print(f'Type of a ==> {type(a)}')
print(f"a ==>  {a}")
```

```
Type of l ==> <class 'list'>
Type of a ==> <class 'numpy.ndarray'>
a ==>  [10 20 30]
```

To know properties of ndarray

- **np.ndim** ==> to know the dimension of the ndarray
- **np.dtype** ==> to know the data type of the elements in the ndarray
- **np.size** ==> to know the total number of elements in the array.

- **np.shape** ==> returns the shape of an array in tuple form

In [11]:

```
# properties of ndarray
print(f'The dimensions of the array a ==> {a.ndim}')
print(f'The data type of elements of array a ==> {a.dtype}')
print(f'The size of the array a ==> {a.size}')
print(f'The shape of the array a ==> {a.shape}')
```

```
The dimensions of the array a ==> 1
The data type of elements of array a ==> int32
The size of the array a ==> 3
The shape of the array a ==> (3,)
```

In [12]:

```
# 1-D array creating using tuples
t = ('krishna','nandu','moneesh')
print(f'Type of t ==> {type(t)}')
a = np.array(t)
print(f'Type of a ==> {type(a)}')
print(f'The data type of elements of array a ==> {a.dtype}')
print(f'The size of the array a ==> {a.size}')
print(f'The shape of the array a ==> {a.shape}')
print(f"a ==> {a}")
```

```
Type of t ==> <class 'tuple'>
Type of a ==> <class 'numpy.ndarray'>
The data type of elements of array a ==> <U7
The size of the array a ==> 3
The shape of the array a ==> (3,)
a ==> ['krishna' 'nandu' 'moneesh']
```

2-D array(Matrix) creation using Nested Lists

In [13]:

```
# [[10,20,30],[40,50,60],[70,80,90]] --->Nested List
a = np.array([[10,20,30],[40,50,60],[70,80,90]])
print(f'Type of a ==> {type(a)}')
print(f'a ==> \n {a}')
```

```
print(f'The dimensions of the array a ==> {a.ndim}')
print(f'The data type of elements of array a ==> {a.dtype}')
print(f'The size of the array a ==> {a.size}')
print(f'The shape of the array a ==> {a.shape}')
```

```
Type of a ==> <class 'numpy.ndarray'>
a ==>
 [[10 20 30]
 [40 50 60]
 [70 80 90]]
The dimensions of the array a ==>  2
The data type of elements of array a ==>  int32
The size of the array a ==> 9
The shape of the array a ==> (3, 3)
```

Note:

- **Array** contains only **homogenous** elements
- If **list** contains **heterogenous elements**, and while creating the array **upcasting** will be performed.

In [14]:

```
# list contains heterogenous elements
l = [10,20,30.5]
a = np.array(l) # upcasting to float
print(f'a :: {a}')
print(f'data type of elements of a ==> {a.dtype}')
```

```
a :: [10.  20.  30.5]
data type of elements of a ==> float64
```

Creating arrays with particular datatype

- we have to use **dtype** argument while creating the arrray

In [15]:

```
# int type
a = np.array([10,20,30.5],dtype=int)
print(a)
```

```
[10 20 30]
```

In [16]:

```
# float type
a = np.array([10,20,30.5],dtype=float)
print(a)
```

```
[10.  20.  30.5]
```

In [17]:

```
# bool type ==> any number is True and zero is False. Any string is True and empty string is False
a = np.array([10,20,30.5],dtype=bool)
print(a)
```

```
[ True  True  True]
```

In [18]:

```
# complex type
a = np.array([10,20,30.5,0],dtype=complex)
print(a)
```

```
[10. +0.j 20. +0.j 30.5+0.j  0. +0.j]
```

In [19]:

```
# str type
a = np.array([10,20,30.5],dtype=str)
print(a)
```

```
['10' '20' '30.5']
```

creating object type array

- **object** is the parent of all int, float, bool, complex and str
- Here the **elements are looks like heterogeneous**. But the datatype of elements is **'object'**

In [20]:

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

# data type
print(f"Data type of elements of array a ==> {a.dtype}")
```

```
[10 'krishna' 10.5 True (10+20j)]
Data type of elements of array a ==> object
```

arange()

- we can **create only 1-D arrays** with **arange()** function

In [21]:

```
import numpy as np
help(np.arange)
```

```
Help on built-in function arange in module numpy:

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

    Return evenly spaced values within a given interval.

    Values are generated within the half-open interval ``[start, stop)``
    (in other words, the interval including `start` but excluding `stop`).
    For integer arguments the function is equivalent to the Python built-in
    `range` function, but returns an ndarray rather than a list.

    When using a non-integer step, such as 0.1, the results will often not
    be consistent.  It is better to use `numpy.linspace` for these cases.
```

Python:

1. range(n) ==> n values from 0 to n-1

- range(4) ==> 0,1,2,3

2. range(m,n)==> from m to n-1

- range(2,7)==> 2,3,4,5,6

3. range(begin,end,step)

- range(1,11,1) ==> 1,2,3,4,5,6,7,8,9,10
- range(1,11,2) ==> 1,3,5,7,9
- range(1,11,3) ==> 1,4,7,10

In [22]:

```
# only stop value ==> 0 to stop-1
import numpy as np
a = np.arange(10)
print(f'a ==> {a}')
print(f'The dimensions of the array a ==> {a.ndim}')
print(f'The data type of elements of array a ==> {a.dtype}')
print(f'The size of the array a ==> {a.size}')
print(f'The shape of the array a ==> {a.shape}')
```

```
a ==> [0 1 2 3 4 5 6 7 8 9]
The dimensions of the array a ==> 1
The data type of elements of array a ==> int32
The size of the array a ==> 10
The shape of the array a ==> (10,)
```

In [23]:

```
# both start and stop values: start to stop-1
import numpy as np
a = np.arange(1,11)
print(f'a ==> {a}')
print(f'The dimensions of the array a ==> {a.ndim}')
print(f'The data type of elements of array a ==> {a.dtype}')
print(f'The size of the array a ==> {a.size}')
print(f'The shape of the array a ==> {a.shape}')
```

```
a ==> [ 1  2  3  4  5  6  7  8  9 10]
The dimensions of the array a ==> 1
The data type of elements of array a ==> int32
The size of the array a ==> 10
The shape of the array a ==> (10,)
```

In [24]:

```
# start, stop and step values : start to stop-1 with step
import numpy as np
a = np.arange(1,11,2)
print(f'a ==> {a}')
print(f'The dimensions of the array a ==> {a.ndim}')
```

```
print(f'The data type of elements of array a ==> {a.dtype}')
print(f'The size of the array a ==> {a.size}')
print(f'The shape of the array a ==> {a.shape}')
```

```
a ==> [1 3 5 7 9]
The dimensions of the array a ==> 1
The data type of elements of array a ==> int32
The size of the array a ==> 5
The shape of the array a ==> (5,)
```

In [25]:

```
# with a particular datatype of the elements
import numpy as np
a = np.arange(1,11,3,dtype=float)
print(f'a ==> {a}')
print(f'The dimensions of the array a ==> {a.ndim}')
print(f'The data type of elements of array a ==> {a.dtype}')
print(f'The size of the array a ==> {a.size}')
print(f'The shape of the array a ==> {a.shape}')
```

```
a ==> [ 1.  4.  7. 10.]
The dimensions of the array a ==> 1
The data type of elements of array a ==> float64
The size of the array a ==> 4
The shape of the array a ==> (4,)
```

linspace()

- same as **arange()** only but in the specified interval, linearly spaced values

In [26]:

```
import numpy as np
help(np.linspace)
```

```
Help on function linspace in module numpy:

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`].
```

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

- Both **start and stop are includded** because **endpoint=True**
- If **endpoint=False** then **stop is excluded**
- **retstep** denotes the spacing between the points.If **True** then the value is returned
- calculation of spacing **(stop-start)/(num-1)** if **endpoint=True**
- calculation of spacing **(stop-start)/(num)** if **endpoint=False**

In [27]:

```
# 50 evenly spaced values between 0 and 1 are returned including both 0 and 1
np.linspace(0,1)
```

Out[27]:

```
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.        ])
```

In [28]:

```
# 4 evenly spaced valued between 0 and 1 including 0 and 1
np.linspace(0,1,4)
```

Out[28]:

```
array([0.        , 0.33333333, 0.66666667, 1.        ])
```

In [29]:

```
# 4 evenly spaced valued between 0 and 1 including 0 and excluding 1
np.linspace(0,1,4,endpoint=False)
```

Out[29]:

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

In [30]:

```
# 4 evenly spaced valued between 0 and 1 including 0 and excluding 1 and return spacing
np.linspace(0,1,4,endpoint=False,retstep=True)
```

Out[30]:

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

In [31]:

```
# 10 values between 1 to 100 including 1 and 100 with equally spaced int values
np.linspace(1,100,10,dtype=int,retstep=True)
```

Out[31]:

```
(array([  1,  12,  23,  34,  45,  56,  67,  78,  89, 100]), 11.0)
```

arange() vs linspace()

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

zeros

- array is filled with 0s

In [32]:

```
import numpy as np
help(np.zeros)
```

```
Help on built-in function zeros in module numpy:

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

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

    Parameters
    ----------
    shape : int or tuple of ints
        Shape of the new array, e.g., ``(2, 3)`` or ``2``.
    dtype : data-type, optional
```

```
        The desired data-type for the array, e.g., `numpy.int8`.  Default is
        `numpy.float64`.
    order : {'C', 'F'}, optional, default: 'C'
        Whether to store multi-dimensional data in row-major
        (C-style) or column-major (Fortran-style) order in
        memory.
```

- **0-D array** ==> Scalar :: single value
- **1-D array** ==> Vector :: Collection of 0-D arrays
- **2-D array** ==> Matrix :: Collection of 1-D arrays
- **3-D array** ==> collcetion of 2-D arrays
- **(10,)** ==>1-D array contains 10 elements
- **(5,2)** ==>2-D array contains 5 rows and 2 columns
- **(2,3,4)** ==>3-D array
  - **2 ==> number of 2-D arrays**
  - **3 ==> The number of rows in every 2-D array**
  - **4 ==> The number of columns in every 2-D array**
  - **size: 2* *3** 4 = 24**

In [33]:

```
# zeros(shape, dtype=float, order='C', *, like=None) ==> shape always in tuple form
# 1-D array with zeros
np.zeros(4)
```

Out[33]:

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

In [34]:

```
# 2-D arrays with zeros
np.zeros((4,3))
```

Out[34]:

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

In [35]:

```
# 3-D array
np.zeros((2,3,4))
```

Out[35]:

```
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.]]])
```

Note: observe the above output.

- outermost there are 3 square brackets are present means it is 3-D array
- exclude one outermost bracket then it is a 2-D array ==> total 2 2-D arrays present
- Each 2-D array contains 1-D arrays. Each 2-D contains 3-rows and 4-columns

In [36]:

```
# 4-D array ==> collection of 3-D arrays
# (2,2,3,4)
# 2 ==> no. of 3-D arrays
# 2 ==> every 3-D 2 2-D arrays
# 3 ==> Every 2-D array contains 3 rows
# 4 ==> every 2-D array contains 4 columns
np.zeros((2,2,3,4))
```

Out[36]:

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

        [[0., 0., 0., 0.],
         [0., 0., 0., 0.],
         [0., 0., 0., 0.]]],
```

```
[[[0., 0., 0., 0.],
  [0., 0., 0., 0.],
  [0., 0., 0., 0.]],

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

ones()

- exactly same as **zeros** except array is filled with 1

In [37]:

```
import numpy as np
help(np.ones)
```

```
Help on function ones in module numpy:

ones(shape, dtype=None, order='C', *, like=None)
    Return a new array of given shape and type, filled with ones.

    Parameters
    ----------
    shape : int or sequence of ints
        Shape of the new array, e.g., ``(2, 3)`` or ``2``.
    dtype : data-type, optional
        The desired data-type for the array, e.g., `numpy.int8`.  Default is
        `numpy.float64`.
    order : {'C', 'F'}, optional, default: C
        Whether to store multi-dimensional data in row-major
        (C-style) or column-major (Fortran-style) order in
        memory.
```

In [38]:

```
# 1-D array
np.ones(10)
```

Out[38]:

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

In [39]:

```
# 2-D array
np.ones((5,2),dtype=int)
```

Out[39]:

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

In [40]:

```
# 3-D array
np.ones((2,3,4),dtype=int)
```

Out[40]:

```
array([[[1, 1, 1, 1],
        [1, 1, 1, 1],
        [1, 1, 1, 1]],

       [[1, 1, 1, 1],
        [1, 1, 1, 1],
        [1, 1, 1, 1]]])
```

full

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

In [41]:

```
import numpy as np
help(np.full)
```

```
Help on function full in module numpy:

full(shape, fill_value, dtype=None, order='C', *, like=None)
    Return a new array of given shape and type, filled with `fill_value`.

    Parameters
    ----------
    shape : int or sequence of ints
        Shape of the new array, e.g., ``(2, 3)`` or ``2``.
```

```
    fill_value : scalar or array_like
        Fill value.
    dtype : data-type, optional
        The desired data-type for the array  The default, None, means
         `np.array(fill_value).dtype`.
    order : {'C', 'F'}, optional
        Whether to store multidimensional data in C- or Fortran-contiguous
        (row- or column-wise) order in memory.
```

In [42]:

```
# 1-D array
np.full(10,fill_value=2)
```

Out[42]:

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

In [43]:

```
# 2-D array
np.full((2,3),fill_value=3)
```

Out[43]:

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

In [44]:

```
# 3-D array
np.full((2,3,4),fill_value=8)
```

Out[44]:

```
array([[[8, 8, 8, 8],
        [8, 8, 8, 8],
        [8, 8, 8, 8]],

       [[8, 8, 8, 8],
        [8, 8, 8, 8],
        [8, 8, 8, 8]]])
```

All of the following will give same results

- np.full(shape=(2,3,4),fill_value=8)
- np.full((2,3,4),fill_value=8)
- np.full((2,3,4),8)

eye()

- To generate **identity matrix**
- Return a **2-D array** with **ones on the diagonal** and **zeros elsewhere.**

In [45]:

```
import numpy as np
help(np.eye)
```

```
Help on function eye in module numpy:

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.
    order : {'C', 'F'}, optional
        Whether the output should be stored in row-major (C-style) or
        column-major (Fortran-style) order in memory.
```

positional arguments only and keyword arguments only

1. **f(a,b)** ==> We can pass positional as well as keyword arguments
2. **f(a,/,b**) ==> before '/' we should pass **positional arguments** only for variables i.e., here for **a**

3. **f(a,*,*b)* ==> *after '*'* we should pass **keyword arguments** only for the variables ie., here for **b**

In [46]:

```
# both positional and keywrod arguments
def f(a,b):
   print(f'The value of a :: {a}')
   print(f'The value of b :: {b}')
f(10,20)
f(a=10,b=20)
```

```
The value of a :: 10
The value of b :: 20
The value of a :: 10
The value of b :: 20
```

In [47]:

```
# position_only.py
def f1(a, /,b):
   print(f'The value of a :{a}')
   print(f'The value of b :{b}')
print("Calling f1(10,20)==> Valid")
f1(10,20)  # valid
print("Calling f1(a=10,b=20) ==> Invalid")
f1(a=10,b=20)
# invalid because before '/' the value for variable should be passed as positional only
```

```
Calling f1(10,20)==> Valid
The value of a :10
The value of b :20
Calling f1(a=10,b=20) ==> Invalid

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-47-743e61686e1a> in <module>
      6 f1(10,20)  # valid
      7 print("Calling f1(a=10,b=20) ==> Invalid")
----> 8 f1(a=10,b=20)
      9 # invalid because befoe '/' the value for variable should be passed a
s positional only
```

```
TypeError: f1() got some positional-only arguments passed as keyword argument
s: 'a'
```

In [48]:

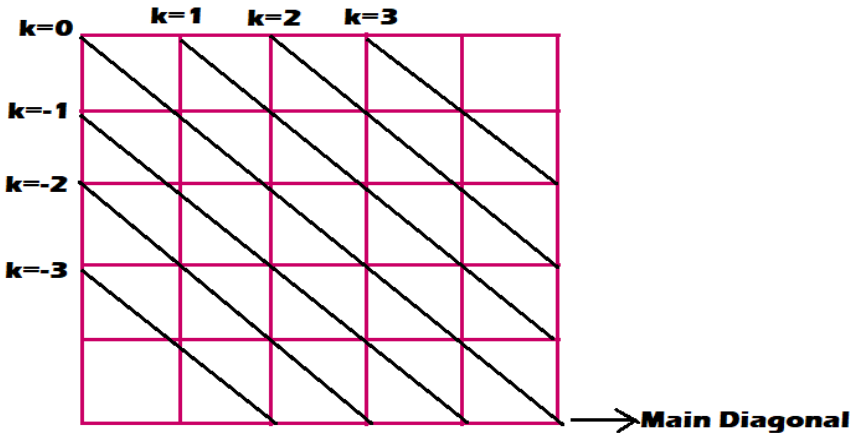
```
def f1(a,*,b):
  print(f'The value of a :: {a}')
  print(f'The value of b :: {b}')
f1(a=10,b=50) #valid
f1(10,20)

# invalid because after '*' we have to provide value for variable(b) through keyword only
```

```
The value of a :: 10
The value of b :: 50

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-48-67b108893ea2> in <module>
      3     print(f'The value of b :: {b}')
      4 f1(a=10,b=50) #valid
----> 5 f1(10,20) # invalid because after '*' we have to provide value for va
riable(b) through keyword only

TypeError: f1() takes 1 positional argument but 2 were given
```

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.



- **N** ==> No of rows
- **M** ==> No. of columns
- **K** ==> Decides in which diagonal the value should be filled with 1s

0 ==> Main diagonal

1 ==> above the main diagonal

-1 ==> below the main diagonal

In [49]:

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

Out[49]:

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

In [50]:

```
np.eye(5,k=1)
```

Out[50]:

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

In [51]:

```
np.eye(5,k=-1)
```

Out[51]:

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

eye() properties

1. It will returns always **2-D arrays**
2. The number of rows and number of columns need not be same.
3. If we omit the 'M' value then the value will be same as 'N'
4. Bydefault main diagonal contains 1s. But we can customize the diagonal which has to contain 1s.

identity()

exactly same as 'eye()' function except

1. It is always square matrix(The number of rows and number of columns always same)
2. only main diagonal contains 1s

In [52]:

```
import numpy as np
help(np.identity)
```

```
Help on function identity in module numpy:

identity(n, dtype=None, *, like=None)
    Return the identity array.
```

In [53]:

```
# identity(n, dtype=None, *, like=None)
np.identity(3,dtype=int)
```

Out[53]:

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

diag()

- **diag(v, k=0)** ==> Extract a diagonal or construct a diagonal array.
- If we provide 2-D array then it will extract the elements at k-th diagonal
- If we provide 1-D array the it will construct a diagonal array with the provided elements and remaining elements are filled with zeros

In [54]:

```
import numpy as np
help(np.diag)
```

```
Help on function diag in module numpy:

diag(v, k=0)
    Extract a diagonal or construct a diagonal array.
```

In [55]:

```
# Extract 2-D diagonal elements
a = np.arange(1,10).reshape(3,3)
print(f"Original 2-D array ==> \n {a}")
print(f"Elements prsent at 0-diagonal ==> {np.diag(a,k=0)}")
print(f"Elements prsent at 1-diagonal ==> {np.diag(a,k=1)}")
print(f"Elements prsent at 2-diagonal ==> {np.diag(a,k=2)}")
print(f"Elements prsent at -1-diagonal ==> {np.diag(a,k=-1)}")
print(f"Elements prsent at -2-diagonal ==> {np.diag(a,k=-2)}")
print(f"Elements prsent at 3-diagonal ==> {np.diag(a,k=3)}")
```

```
Original 2-D array ==>
 [[1 2 3]
 [4 5 6]
 [7 8 9]]
Elements prsent at 0-diagonal ==> [1 5 9]
Elements prsent at 1-diagonal ==> [2 6]
Elements prsent at 2-diagonal ==> [3]
Elements prsent at -1-diagonal ==> [4 8]
```

```
Elements prsent at -2-diagonal ==> [7]
Elements prsent at 3-diagonal ==> []
```

In [56]:

```
# 1-D construct 2-D array with the diagonal array with the provided elements
# remaining elements are filled with zeros
a = np.array([10,20,30,40])
np.diag(a,k=0)
```

Out[56]:

```
array([[10,  0,  0,  0],
       [ 0, 20,  0,  0],
       [ 0,  0, 30,  0],
       [ 0,  0,  0, 40]])
```

In [57]:

```
np.diag(a,k=1)
```

Out[57]:

```
array([[ 0, 10,  0,  0,  0],
       [ 0,  0, 20,  0,  0],
       [ 0,  0,  0, 30,  0],
       [ 0,  0,  0,  0, 40],
       [ 0,  0,  0,  0,  0]])
```

In [58]:

```
np.diag(a,k=-1)
```

Out[58]:

```
array([[ 0,  0,  0,  0,  0],
       [10,  0,  0,  0,  0],
       [ 0, 20,  0,  0,  0],
       [ 0,  0, 30,  0,  0],
       [ 0,  0,  0, 40,  0]])
```

empty()

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

In [59]:

```
import numpy as np
help(np.empty)
```

```
Help on built-in function empty in module numpy:

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

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

In [60]:

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

Out[60]:

```
array([[0.00000000e+000, 0.00000000e+000, 0.00000000e+000],
       [0.00000000e+000, 0.00000000e+000, 6.42285340e-321],
       [4.47593775e-091, 4.47593775e-091, 1.06644825e+092]])
```

Note:

- There is no default value in C language
- So if value is initialized then it will be assigned with some garbage values
- Generally 'empty()' function is ued to create dummy array.
- np.empty(10) returns uninitialized data ==> we can use this for future purpose

zeros() vs empty()

1. If we required an array only with zeros then we should go for zeros().
2. If we never worry about data, just we required an empty array for future purpose, then we should go for empty().
3. 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.

In [61]:

```
# performance comaparision of zeros() and empty()
import numpy as np
from datetime import datetime
import sys

begin = datetime.now()
a = np.zeros((10000,300,400))
after = datetime.now()
print('Time taken by zeros:',after-begin)

a= None
begin = datetime.now()
a = np.empty((10000,300,400))
after = datetime.now()
print('Time taken by empty:',after-begin)
```

```
Time taken by zeros: 0:00:00.027922
Time taken by empty: 0:00:00.028924
```

Note:

- **Numpy** : basic datatype ==> ndarray
- **Scipy** : Advance Numpy
- **Pandas** : basic datatypes => Series(1-D) and DataFrame(Multi dimensional) based on Numpy
- **Matplotlib** : Data visualization module
- **Seaborn, plotly** : based on Matplotlib

np.random module

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

randint

- To generate **random int** values in the given range

In [62]:

```
import numpy as np
help(np.random.randint)
```

```
Help on built-in function randint:

randint(...) method of numpy.random.mtrand.RandomState instance
    randint(low, high=None, size=None, dtype=int)

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

    Return random integers from the "discrete uniform" distribution of
    the specified dtype in the "half-open" interval [`low`, `high`). If
    `high` is None (the default), then results are from [0, `low`).
```

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

- Return random integers from `low` (inclusive) to `high` (exclusive).
- it is represented as **[low, high)** ==> **[means inclusive** and **) means exclusive**
- If **high is None** (the default), then results are from **[0, low)**

In [63]:

```
# it will generate a single random int value in the range 10 to 19.
np.random.randint(10,20)
```

Out[63]:

```
10
```

In [64]:

```
# To create 1-D ndarray of size 10 with random values from 1 to 8?
np.random.randint(1,9,size=10)
```

Out[64]:

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

In [65]:

```
# To create 2D array with shape(3,5)
np.random.randint(100,size=(3,5))
```

Out[65]:

```
array([[98, 32, 45, 60,  7],
       [97, 42, 71,  0, 17],
       [17, 44,  2, 29, 45]])
```

In [66]:

```
# from 0 to 99 random values, 2-D array will be created. Here high is None
np.random.randint(100,size=(3,5))
```

Out[66]:

```
array([[92, 29, 24, 20, 56],
       [12, 20, 54, 83, 12],
       [ 5, 17, 55, 60, 62]])
```

In [67]:

```
# To create 3D array with shape(2,3,4)
# 3D array contains 2  2-D arrays
# Each 2-D array contains 3 rows and 4 columns
# np.random.randint(100,size=(2,3,4))
np.random.randint(100,size=(2,3,4))
```

Out[67]:

```
array([[[35,  1, 54, 25],
        [ 5, 44, 93, 60],
        [27, 24, 23, 51]],

       [[65, 55, 22, 74],
        [76, 19,  9, 43],
        [17, 44, 58, 52]]])
```

In [68]:

```
# randint(low, high=None, size=None, dtype=int)
# dtype must be int types
# int types
# int8
# int16
# int32 (default)
# int64
```

```
a = np.random.randint(1,11,size=(20,30))
print(f"Data type of the elements : {a.dtype}")
```

```
Data type of the elements : int32
```

```
In [69]:

# memory utilization is improved with dtype
import sys
a = np.random.randint(1,11,size=(20,30))
print(f"with int32 size of ndarray : {sys.getsizeof(a)}")
a = np.random.randint(1,11,size=(20,30),dtype='int8')
print(f"with int8 size of ndarray : {sys.getsizeof(a)}")
```

```
with int32 size of ndarray : 2520
with int8 size of ndarray : 720
```

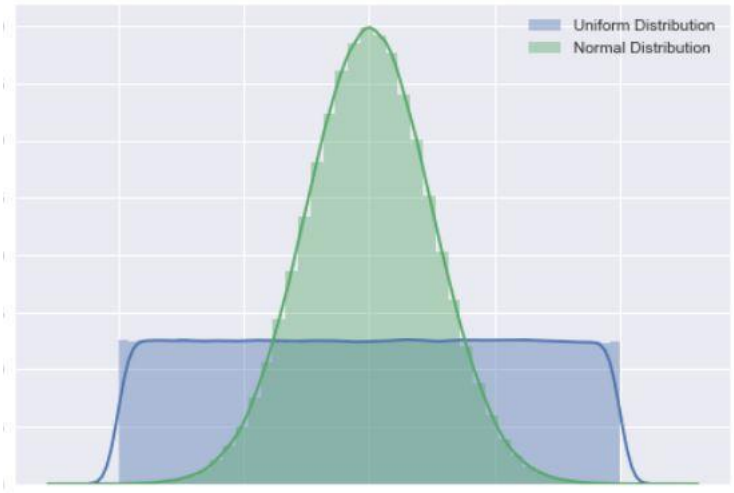
astype() ==> To convert the data type from one form to another

```
In [70]:

# we can convert the datatype to another.
# by default randint() will return int datatype.
# by using "astype()" we can convert one datatype to another
a = np.random.randint(1,11,size=(20,30))
print("Before Conversion ")
print(f"Data type of a ==> {a.dtype}")
b = a.astype('float')
print("Ater calling astype() on the ndarray a ")
print(f"Data type of b==> {b.dtype}")
```

```
Before Conversion
Data type of a ==> int32
Ater calling astype() on the ndarray a
Data type of b==> 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.



rand()

- It will generates random float values in the range [0,1) from uniform distribution samples.
- **[0 ==> means 0 is incluede and 1) ==> means 1 is excluded**

In [71]:

```
import numpy as np
help(np.random.rand)
```

```
Help on built-in function rand:

rand(...) method of numpy.random.mtrand.RandomState instance
    rand(d0, d1, ..., dn)

    Random values in a given shape.
```

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

Random values in a given shape.

In [72]:

```
# single random float value from 0 to 1
np.random.rand()
```

Out[72]:

```
0.08638507965381459
```

In [73]:

```
# 1-D array with float values from 0 to 1
np.random.rand(10)
```

Out[73]:

```
array([0.39130964, 0.4701559 , 0.49176215, 0.38386718, 0.35090334,
       0.02928118, 0.96359654, 0.41079371, 0.38436428, 0.81939376])
```

In [74]:

```
# 2-D array with float values from 0 to 1
np.random.rand(3,5)
```

Out[74]:

```
array([[0.64607105, 0.92823898, 0.48098258, 0.52641539, 0.09602147],
       [0.36884988, 0.29296605, 0.87343336, 0.67168146, 0.09297364],
       [0.21607014, 0.14382148, 0.47534017, 0.84083409, 0.73185496]])
```

In [75]:

```
# 3-D array with float values from 0 to 1
np.random.rand(2,3,4)
```

Out[75]:

```
array([[[0.83901492, 0.03543901, 0.76098031, 0.46620334],
        [0.25545172, 0.24279657, 0.66570238, 0.34390092],
        [0.44146884, 0.04426514, 0.59433418, 0.25362922]],

       [[0.88437233, 0.04283568, 0.57814391, 0.91268089],
        [0.72943145, 0.95593275, 0.26450772, 0.75816229],
        [0.74559404, 0.22803979, 0.34306227, 0.33591768]]])
```

uniform()

- **rand()** ==> range is always [0,1)
- **uniform()** ==> customize range [low,high)

In [76]:

```
import numpy as np
help(np.random.uniform)
```

```
Help on built-in function uniform:

uniform(...) method of numpy.random.mtrand.RandomState instance
    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).  In other words,
    any value within the given interval is equally likely to be drawn
    by `uniform`.
```

In [77]:

```
# uniform(low=0.0, high=1.0, size=None)
np.random.uniform()   # it will acts same as np.random.rand()
```

Out[77]:

```
0.8633674116602018
```

In [78]:

```
# random float value in the given customized range
np.random.uniform(10,20)
```

Out[78]:

```
10.297706247279303
```

In [79]:

```
# 1-D array with customized range of float values
np.random.uniform(10,20,size=10)
```

Out[79]:

```
array([16.25675741, 10.57357925, 18.09157687, 19.07874982, 17.06463829,
       11.98365064, 12.01449467, 10.8823314 , 18.03513355, 17.66067279])
```

In [80]:

```
# 2-D array with customized range of float values
np.random.uniform(10,20,size=(3,5))
```

Out[80]:

```
array([[16.96942587, 15.68977979, 16.09812119, 13.06668784, 15.36258784],
       [14.55135047, 10.27434721, 19.46874406, 16.33163335, 19.07160274],
       [15.1368871 , 18.83658294, 10.66735409, 14.30008464, 13.79529403]])
```

In [81]:

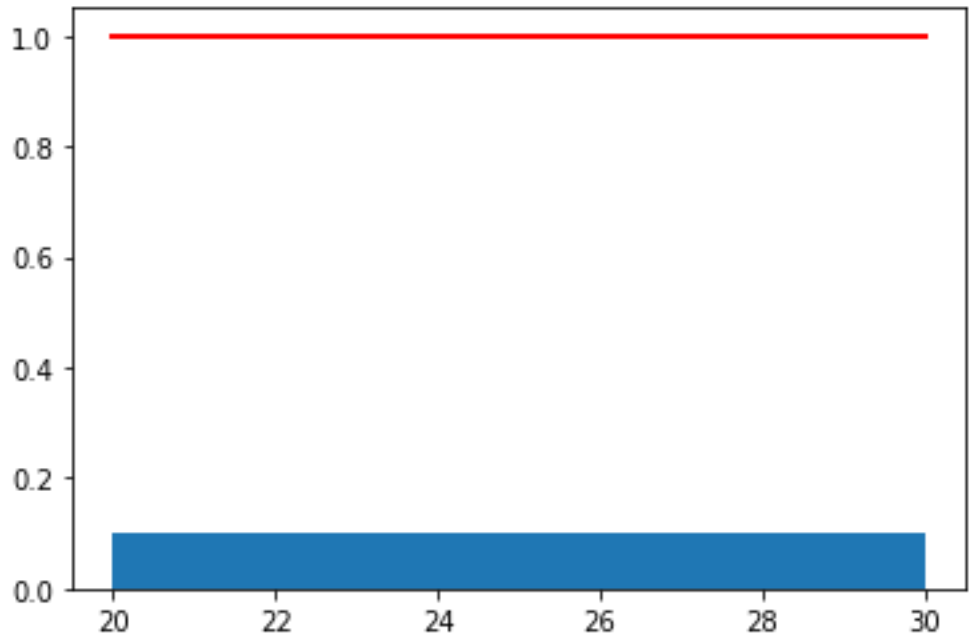
```
# 3-D array with customized range of float values
np.random.uniform(10,20,size=(2,3,4))
```

Out[81]:

```
array([[[19.71747034, 13.13603927, 13.55533583, 13.49569866],
        [18.18364556, 11.64037815, 13.12254598, 10.73172933],
        [18.44982662, 11.48966867, 15.66803442, 12.37854234]],

       [[16.01656014, 12.22451809, 18.524565  , 17.29353028],
        [15.33632839, 17.41720778, 17.3426583 , 12.98066622],
        [17.02012869, 19.74650958, 15.33698393, 16.86296185]]])
```

In [82]:

```
# to demonstrate the Unifrom distribution in data visualization
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()

- values from **normal distribution** with mean 0 and varience(stadard deviation) is 1

In [83]:

```
import numpy as np
help(np.random.randn)
```

```
Help on built-in function randn:

randn(...) method of numpy.random.mtrand.RandomState instance
    randn(d0, d1, ..., dn)

    Return a sample (or samples) from the "standard normal" distribution.
```

In [84]:

```
# randn(d0, d1, ..., dn)
# single float value  including -ve value also
a = np.random.randn()
print(a)
```

```
-0.0563667809620408
```

In [85]:

```
# 1-D array with mean 0 and stand deviation 1
a = np.random.randn(10)
print(a)
print(f"Mean ==> {a.mean()}")
print(f"Variance ==> {a.var()}")
print(f"Standard deviation ==> {a.std()}")
```

```
[-2.59949374  1.07020758 -1.525435    0.97692928  0.71398471 -0.08671092
  0.07833857  0.70247123  0.94548715  0.38984236]
Mean ==> 0.06656212089297449
Variance ==> 1.3202568278489202
Standard deviation ==> 1.1490242938462703
```

In [86]:

```
# 2-D array with float values in normal distribution
np.random.randn(2,3)
```

Out[86]:

```
array([[-1.84491706e+00, -1.80823278e-03,  1.61444938e-01],
       [-1.05642111e-01,  1.38334944e+00, -8.11945928e-01]])
```

In [87]:

```
# 3-D array with float values in normal distribution
np.random.randn(2,3,4)
```

Out[87]:

```
array([[[-0.67500418, -0.04050421, -1.28250359, -0.22205388],
        [-0.82973108,  0.69627796, -0.44151688, -1.05482818],
        [-0.39583573, -0.45231744,  0.72255689, -2.02508302]],

       [[ 0.56678912, -0.07604355, -0.62011088, -1.57825963],
        [-0.51442293,  0.52223471,  0.44212152,  0.85563198],
        [-1.17712106,  0.06659177, -2.06951363, -0.39449518]]])
```

- **randint()** ==> To generate random int values in the given range.
- **rand()** ==> uniform distribution in the range [0,1)
- **uniform()** ==> uniform distribution in our provided range
- **randn()** ==> normal distribution values with mean 0 and variance is 1

- **normal()** ==> normal distribution values with our mean and variance.

normal()

- We can customize mean and variance

In [88]:

```
import numpy as np
help(np.random.normal)
```

```
Help on built-in function normal:

normal(...) method of numpy.random.mtrand.RandomState instance
    normal(loc=0.0, scale=1.0, size=None)

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

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

- **loc** : float or array_like of floats
  Mean ("centre") of the distribution.
- **scale** : float or array_like of floats
  Standard deviation (spread or "width") of the distribution. Must be non-negative.
- **size** : int or tuple of ints, optional

In [89]:

```
# equivalent ot np.random.randn()
np.random.normal()
```

Out[89]:

```
-1.5337940293701702
```

In [90]:

```
# 1-D array with float values in normal distribution
a = np.random.normal(10,4,size=10)
print(f" a ==>  {a}")
print(f"Mean ==>  {a.mean()}")
print(f"Variance ==>  {a.var()}")
print(f"Standard deviation ==> {a.std()}")
```

```
 a ==>  [12.58568061 12.65559388  2.05077827 11.47857117  8.28240666

        5.81919127  13.26960627  7.08689804 14.17971283 12.81610858]
Mean ==>  10.022454758081924
Variance ==>  14.411542333115623
Standard deviation ==> 3.7962537234905183
```

In [91]:

```
# 2-D array with float values in normal distribution
np.random.normal(10,4,size=(3,5))
```

Out[91]:

```
array([[ 9.60636406, 10.55287084, 13.58021522,  8.25985296, 12.12854933],
       [ 8.93107829,  3.37094761,  8.64304598, 11.32689884, 20.16374935],
       [ 7.09621692, 10.86554327,  5.52023479,  9.4032508 ,  9.75254932]])
```

In [92]:

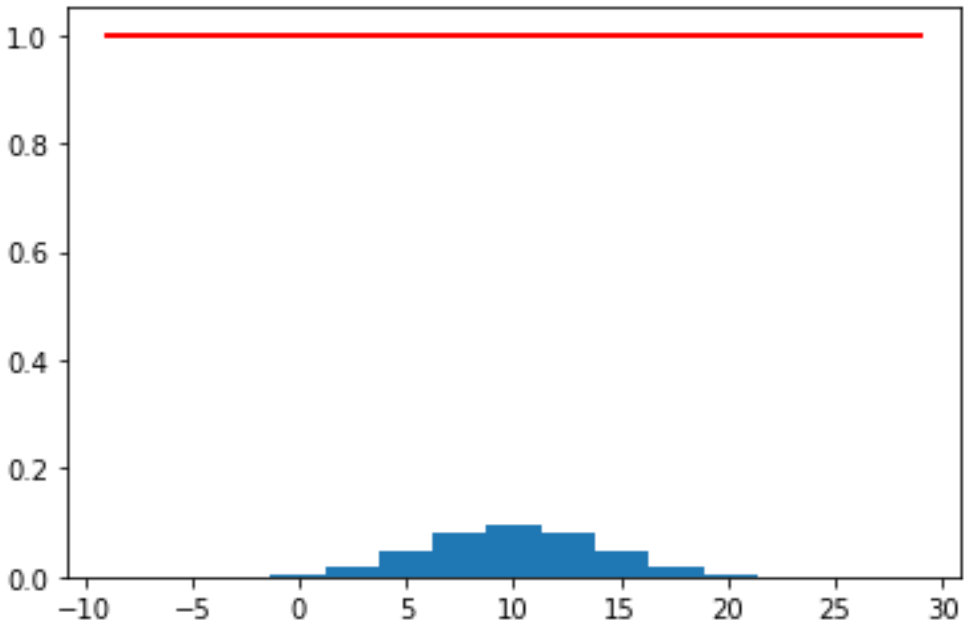
```
# 3-D array with float values in normal distribution
np.random.normal(10,4,size=(2,3,4))
```

Out[92]:

```
array([[[13.00647353, 11.55193581,  7.71907516,  8.66954042],
        [ 6.94062911,  9.64218428,  9.57700968, 11.82765693],
        [16.17363874,  6.88623436,  8.78229986,  5.41236423]],

       [[13.34349311, 14.04491431,  0.86900938,  9.72759205],
        [ 8.67789112,  7.02952354,  9.19535948, 15.08727821],
        [ 3.05669971, 12.80722846, 15.10007439, 14.21390539]]])
```

In [93]:

```
# to demonstrate the Unifrom distribution in data visualization
s = np.random.normal(10,4,1000000)
import matplotlib.pyplot as plt
count, bins, ignored = plt.hist(s, 15, density=True)
plt.plot(bins, np.ones_like(bins), linewidth=2, color='r')
plt.show()
```



shuffle()

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

In [94]:

```
import numpy as np
help(np.random.shuffle)
```

```
Help on built-in function shuffle:

shuffle(...) method of numpy.random.mtrand.RandomState instance
    shuffle(x)

    Modify a sequence in-place by shuffling its contents.

    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.
```

In [95]:

```
# 1-D array
a = np.arange(9)
print(f'a before shuffling : {a}')
np.random.shuffle(a) # inline shuffling happens
print(f'a after shuffling : {a}')
```

```
a before shuffling : [0 1 2 3 4 5 6 7 8]
a after shuffling : [0 6 1 8 4 2 5 3 7]
```

In [96]:

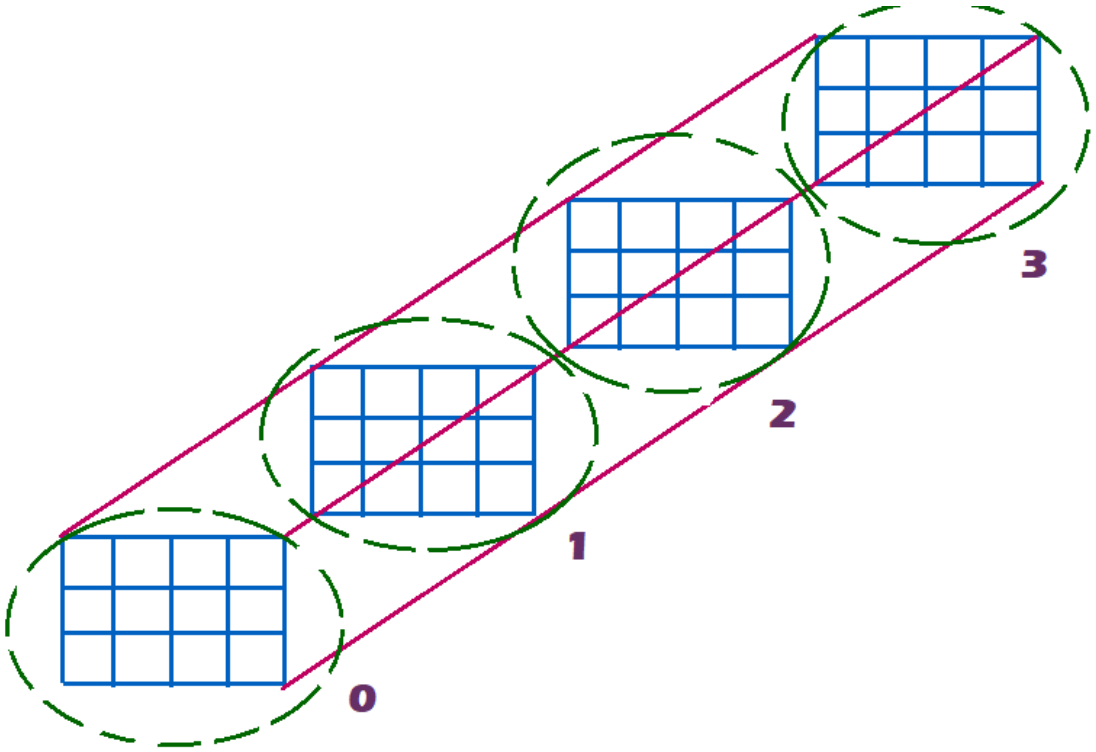
```
# 2-D array
# (6,5)
# axis-0 : number of rows(6)
# axis-1 : number of columns(5)
# This function only shuffles the array along the first axis of a multi-dimensional array(axis-0).
# The order of sub-arrays is changed but their contents remains the same.
a =  np.random.randint(1,101,size=(6,5))
print(f'a before shuffling ==> \n {a}')
np.random.shuffle(a) # inline shuffling happens
print(f'a after shuffling ==>\n {a}')
```

```
a before shuffling ==>
 [[ 26   4  63  66  37]
 [ 65  85  14  51  68]
 [ 59   3  22  25  50]
 [  4  42  40  81  79]
 [ 50  83  70  13   2]
 [100  54  95  39  67]]
a after shuffling ==>
 [[ 59   3  22  25  50]
 [ 65  85  14  51  68]
 [100  54  95  39  67]
 [  4  42  40  81  79]
 [ 26   4  63  66  37]
 [ 50  83  70  13   2]]
```

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

- **4** : number of 2-D arrays (axis-0) ==> Vertical
- **3** : rows in every 2-D array(axis-1) ==> Horizontal
- **4** : columns in every 2-D array(axis-1)

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



In [97]:

```
a = np.arange(48).reshape(4,3,4)
print(f'Before shuffling 3-D array \n :{a}')
np.random.shuffle(a)
print(f'After shuffling 3-D array \n :{a}')
```

```
Before shuffling 3-D 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]]]
After shuffling 3-D array
 :[[[36 37 38 39]
  [40 41 42 43]
  [44 45 46 47]]

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

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

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

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.[low,high)
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.

Chapter-3
Array Attributes and Numpy Data Types

Array Attributes

- **ndim** ==> returns the dimension of the array
- **shape** ==> returns the shape of the array. Always return in tuple form. (10,) : 1-D (10,3): 2-D
- **size** ==> To get total number of elements in the array
- **dtype** ==> To get data type of elements of the array
- **itemsize** ==> Length of each element of array in bytes

In [98]:

```
# 0-D array
import numpy as np
a = np.array(10)
print(f'Dimension of the array a ==> {a.ndim}')
print(f'Shape of the array a ==> {a.shape}')
print(f'Size of the array a ==> {a.size}')
print(f'Data type of the elements in the array a ==> {a.dtype}')
print(f'Itemsize oof the array a ==> {a.itemsize}')
```

```
Dimension of the array a ==> 0
Shape of the array a ==> ()
Size of the array a ==> 1
Data type of the elements in the array a ==> int32
Itemsize oof the array a ==> 4
```

In [99]:

```
# 1-D array
a = np.array([10,20,30])
print(f'Dimension of the array a ==> {a.ndim}')
print(f'Shape of the array a ==> {a.shape}')
print(f'Size of the array a ==> {a.size}')
print(f'Data type of the elements in the array a ==> {a.dtype}')
print(f'Itemsize oof the array a ==> {a.itemsize}')
```

```
Dimension of the array a ==> 1
Shape of the array a ==> (3,)
```

```
Size of the array a ==> 3
Data type of the elements in the array a ==> int32
Itemsize oof the array a ==> 4
```

In [100]:

```
# 2-D array with int32(default)
a = np.array([[10,20,30],[40,50,60],[70,80,90]])
print(f'Dimension of the array a ==> {a.ndim}')
print(f'Shape of the array a ==> {a.shape}')
print(f'Size of the array a ==> {a.size}')
print(f'Data type of the elements in the array a ==> {a.dtype}')
print(f'Itemsize oof the array a ==> {a.itemsize}')
```

```
Dimension of the array a ==> 2
Shape of the array a ==> (3, 3)
Size of the array a ==> 9
Data type of the elements in the array a ==> int32
Itemsize oof the array a ==> 4
```

In [101]:

```
# 2-D array with float64
a = np.array([[10,20,30],[40,50,60],[70,80,90]],dtype='float')
print(f'Dimension of the array a ==> {a.ndim}')
print(f'Shape of the array a ==> {a.shape}')
print(f'Size of the array a ==> {a.size}')
print(f'Data type of the elements in the array a ==> {a.dtype}')
print(f'Itemsize oof the array a ==> {a.itemsize}')
```

```
Dimension of the array a ==> 2
Shape of the array a ==> (3, 3)
Size of the array a ==> 9
Data type of the elements in the array a ==> float64
Itemsize oof the array a ==> 8
```

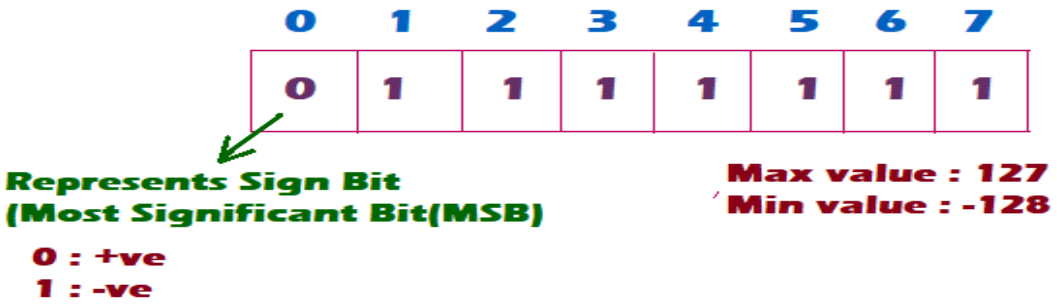
Numpy Data Types

- **Python data types :** int, float, complex, bool and str
- **Numpy data types :** Multiple data types present (Python + C)

- **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** ==> i1 ; **int16** ==> i2; **int32** ==> i4(default)
- **float16** ==> f2 ; **float32** ==> f4(default) ; **float64** ==> f8

int8(i1)

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



int16(i2)

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

int32 (default) (i4)

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

int64

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

In [102]:

```
# For efficient usage of memory we will consider the datatypes
import sys
a = np.array([10,20,30,40])  # default datatype : int32
print(f"Size of int32 : {sys.getsizeof(a)}")
a = np.array([10,20,30,40],dtype='int8')
print(f"Size of int8 : {sys.getsizeof(a)}")
```

```
Size of int32 : 120
Size of int8 : 108
```

Changing the data type of an existing array ==> astype()

we can change the data type of an existing array in two ways

- by using **astype()**
- by using built-in function of numpy like **float64()**

In [103]:

```
# by using 'astype()'
a = np.array([10,20,30,40])
b = a.astype('float64')
print(f"datatype of elements of array a : {a.dtype}")
print(f"datatype of elements of array b : {b.dtype}")
print(a)
print(b)
```

```
datatype of elements of array a : int32
datatype of elements of array b : float64
[10 20 30 40]
[10. 20. 30. 40.]
```

In [104]:

```
# by using built-in function of numpy like float64()
a = np.array([10,20,30,40])
```

```
b = np.float64(a)
print(f"datatype of elements of array a : {a.dtype}")
print(f"datatype of elements of array b : {b.dtype}")
print(a)
print(b)
```

```
datatype of elements of array a : int32
datatype of elements of array b : float64
[10 20 30 40]
[10. 20. 30. 40.]
```

In [105]:

```
# converting to bool type

a = np.array([10,0,20,0,30])
x = np.bool(a)
```

```
<ipython-input-105-93d254e35467>:4: DeprecationWarning: `np.bool` is a deprec
ated alias for the builtin `bool`. To silence this warning, use `bool` by its
elf. Doing this will not modify any behavior and is safe. If you specifically
wanted the numpy scalar type, use `np.bool_` here.
Deprecated in NumPy 1.20; for more details and guidance: https://numpy.org/de
vdocs/release/1.20.0-notes.html#deprecations
  x = np.bool(a)

---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-105-93d254e35467> in <module>
      2
      3 a = np.array([10,0,20,0,30])
----> 4 x = np.bool(a)

ValueError: The truth value of an array with more than one element is ambiguo
us. Use a.any() or a.all()
```

In [106]:

```
x = np.bool_(a)
x
```

Out[106]:

```
array([ True, False,  True, False,  True])
```

Chapter-4
View Vs Copy

View vs Copy

View

- **View** is not separate object and just it is **logical representation of existing array.*8
- If we perform any changes to the original array, those changes will be reflected to the view. vice-versa also.
- We can create view explicitly by using **view()** method of ndarray class.

```
In [107]:

import numpy as np
a = np.array([10,20,30,40])
b = a.view()
print(f"Original Array(a)  ==>  {a}")
print(f"View of the  Array(b)  ==> {b}")
print("*"*80)
print("Changing the values of a, it will reflect to view(b) also :")
a[0]=7777
print(f"Changed Array(a) ==> {a}")
print(f"View of the  Array(b)  ==> {b}")
print("*"*80)
print("Changing the values of view(b), it will reflect to a also :")
b[-1]=999
print(f"Changed view Array(b)  ==> {b}")
print(f"Original Array(a)  ==> {a}")
```

```
Original Array(a)  ==>  [10 20 30 40]
View of the  Array(b)  ==> [10 20 30 40]
********************************************************************************
Changing the values of a, it will reflect to view(b) also :
Changed Array(a) ==> [7777   20   30   40]
View of the  Array(b)  ==> [7777   20   30   40]
********************************************************************************
Changing the values of view(b), it will reflect to a also :
Changed view Array(b)  ==> [7777   20   30  999]
Original Array(a)  ==> [7777   20   30  999]
```

copy

- **Copy** means **separate object.**
- If we perform any changes to the original array, those changes won't be reflected to the Copy. Vice-versa also.
- By using **copy()** method of ndarray class, we can create copy of existing ndarray.

```
In [108]:

import numpy as np

a = np.array([10,20,30,40])
b = a.copy()
print(f"Original Array(a)  ==>  {a}")
print(f"copy of the  Array(b)  ==> {b}")
print("*"*60)

print("Changing the values of a, it will not reflect to copy(b) ")
a[0]=7777
print(f"Changed Array(a) ==> {a}")
print(f"copy of the  Array(b)  ==> {b}")
print("*"*60)

print("Changing the values of copy(b), it will not reflect to a  ")
b[-1]=999
print(f"Changed copy Array(b)  ==> {b}")
print(f"Original Array(a)  ==> {a}")
```

```
Original Array(a)  ==>  [10 20 30 40]
copy of the  Array(b)  ==> [10 20 30 40]
************************************************************
Changing the values of a, it will not reflect to copy(b)
Changed Array(a) ==> [7777   20   30   40]
copy of the  Array(b)  ==> [10 20 30 40]
************************************************************
Changing the values of copy(b), it will not reflect to a
Changed copy Array(b)  ==> [ 10  20  30 999]
Original Array(a)  ==> [7777   20   30   40]
```

Chapter-5
Indexing, Slicing and Advanced Indexing

Accessing Elements of ndarray

1. **Indexing** ==> only one element
2. **Slicing** ==> group of elements which are in order
3. **Advanced indexing** ==> group of elements which are not in order(arbitrary elements)
4. **Condition based selection** ==> array_name[condition]

Indexing -- only one element

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

Accessing elements of 1-D array

- 1-D array : array[index]

Syntax

- **1-D array** :array[index]
- **2-D array** :array[row_index][column_index]
- **3-D array** :array[2-D array index][row_index_in that 2-D array][column*index* in that 2-d array]

In [109]:

```
# 1-D array
# array[index]
import numpy as np
a = np.array([10,20,30,40,50])
a[3]
```

Out[109]:

```
40
```

In [110]:

```
# -ve indexing
a[-1]
```

Out[110]:

```
50
```

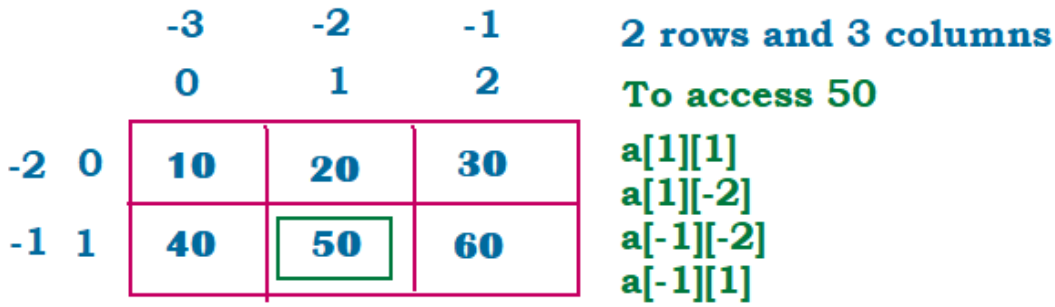
Accessing elements of 2-D array

- 2-D array ==> Collection of 1-D arrays
- There are 2 axis are present in 2-D array

axis-0 :: row index

axis-1 :: column index

- **2-D array : array[row_index][column_index]**



In [111]:

```
# 2-D array
# a[rowindex][columnindex]
import numpy as np
a = np.array([[10,20,30],[40,50,60]])
print(f"Shape of the array a : {a.shape}")
print("To Access the element 50")
print(f"a[1][1] ==> {a[1][1]}")
print(f"a[1][-2] ==> {a[1][-2]}")
print(f"a[-1][2] ==> {a[-1][-2]}")
print(f"a[-1][1] ==> {a[-1][1]}")
```

```
Shape of the array a : (2, 3)
To Access the element 50
a[1][1] ==> 50
a[1][-2] ==> 50
a[-1][2] ==> 50
a[-1][1] ==> 50
```

Accessing elements of 3-D array

- 3-D arrays ==> Collection of 2-D arrays
- There are 3 index in 3-D array

axis-0 :: No of 2-D arrays

axis-1 :: Row index

axis-2 :: column index

- **3-D array : array[2-D array index][row_index_in that 2-D array][column*index* in that 2-d array]**

a[i][j][k]

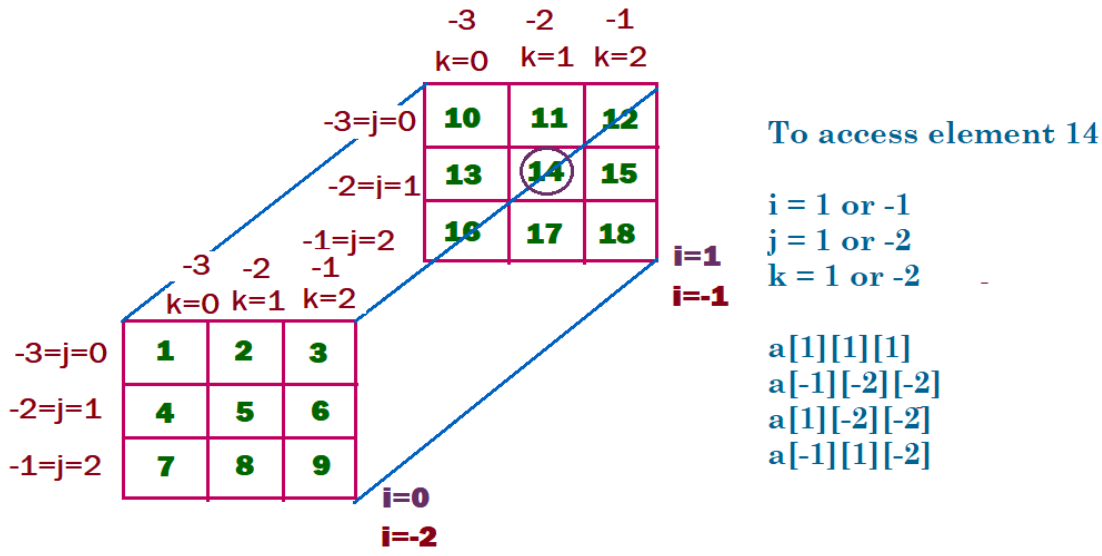
- i ==> represents which 2-D array(index of 2-D array) | can be either +ve or -ve
- j ==> represents row index in that 2-D array | can be either +ve or -ve
- k ==> represents column index in that 2-D array | can be either +ve or -ve

Eg: a[0][1][2]

- 0 indexed 2-D array
- In that 2-D array row-index 1 and column-index 2 will be selected

In [112]:

```
# 3-D array
# array[2-D array index][row_index in that 2-D array][column_index in that 2-D array]
```



To access element 14

i = 1 or -1
j = 1 or -2
k = 1 or -2

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

In [113]:

```
import numpy as np
l = [[[1,2,3],[4,5,6],[7,8,9]],[[10,11,12],[13,14,15],[16,17,18]]]
a = np.array(l)
print(f"Shape of the array a ==> {a.shape}")
print("To access the element 14 from the 3-D array")
print(f"a[1][1][1] ==> {a[1][1][1]}")
print(f"a[-1][-2][-2] ==> {a[-1][-2][-2]}")
print(f"a[1][-2][-2] ==> {a[1][-2][-2]}")
print(f"a[-1][1][-2] ==> {a[-1][1][-2]}")
```

```
Shape of the array a ==> (2, 3, 3)
To access the element 14 from the 3-D array
a[1][1][1] ==> 14
a[-1][-2][-2] ==> 14
a[1][-2][-2] ==> 14
a[-1][1][-2] ==> 14
```

Accessing elements of 4-D array

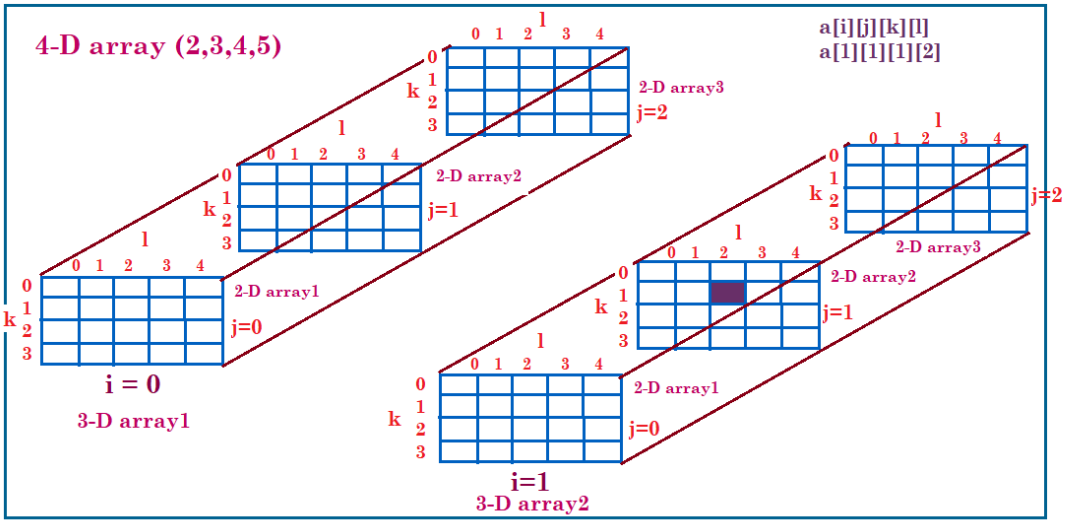
- 4-D array contains multiple 3-D arrays
- Every 3-D array contains multiple 2-D arrays
- Every 2-D array contains rows and columns

(i,j,k,l) ==> **(2,3,4,5)**

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

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

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



In [114]:

```
# 4-D array
import numpy as np
a = np.arange(1,121).reshape(2,3,4,5)
print(f"Size of the array ==> {a.size}")
print(f"Shape of the array ==> {a.shape}")
print(f"a ==>\n {a}")
```

```
Size of the array ==> 120
Shape of the array ==> (2, 3, 4, 5)
a ==>
 [[[[  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  49  50]
   [ 51  52  53  54  55]
   [ 56  57  58  59  60]]]

 [[[ 61  62  63  64  65]
   [ 66  67  68  69  70]
   [ 71  72  73  74  75]
   [ 76  77  78  79  80]]

  [[ 81  82  83  84  85]
   [ 86  87  88  89  90]
   [ 91  92  93  94  95]
   [ 96  97  98  99 100]]

  [[101 102 103 104 105]
   [106 107 108 109 110]
   [111 112 113 114 115]
   [116 117 118 119 120]]]]
```

In [115]:

```
# to know the position of an element in the array
np.where(a==88)
```

Out[115]:

```
(array([1], dtype=int64),
 array([1], dtype=int64),
 array([1], dtype=int64),
 array([2], dtype=int64))
```

In [116]:

```
# accessing the element at a[1][1][1][2]
print(f"The element present at a[1][1][1][2] ==> {a[1][1][1][2]}")
```

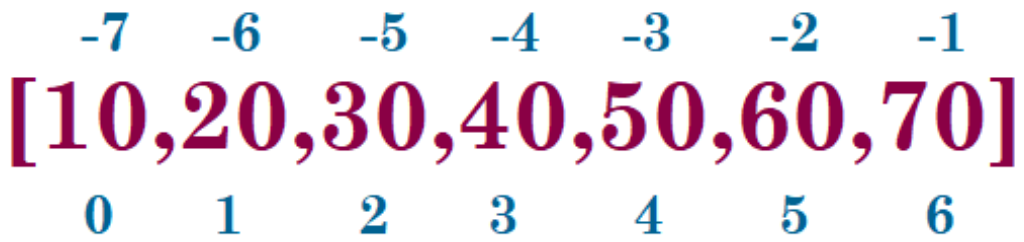
```
The element present at a[1][1][1][2] ==> 88
```

Slicing ==> group of elements which are in order

Python's slice operator:

Syntax-1: l[begin:end]

- It returns elements from begin index to (end-1) index



In [117]:

```
l = [10,20,30,40,50,60,70]
l[2:6] #from 2nd index to 5th index
```

Out[117]:

```
[30, 40, 50, 60]
```

In [118]:

```
l[-6:-2] #From -6 index to -3 index
```

Out[118]:

```
[20, 30, 40, 50]
```

In [119]:

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

Out[119]:

```
[10, 20, 30]
```

In [120]:

```
# If we are not specifying end index then default is upto last
l[2:] # from 2nd index to last
```

Out[120]:

```
[30, 40, 50, 60, 70]
```

In [121]:

```
# if we are not specifying begin and end then it will considered all elements
l[:] # it is equivalent to l
```

Out[121]:

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

Syntax-2: l[begin:end:step]

- **begin,end and step** values can be **either +ve or -ve**. These are optional
- If **begin** is not specified then **0** will be taken **by default**
- If **end** is not specified then **upto end** of the list will be taken
- If **step** is not specified then **1** will be taken **by default**
- **step value cannot be 0**
- If **step value is +ve**, then in **forward direction begin to end-1** will be considered
- If **step value is -ve**, then in **backward direction begin to end+1** will be considered
- In **forward direction**, if **end is 0** then the result will be always **empty**
- In **backward direction**, if **end is -1** then the result will be always **empty**

Note:

- slice operator **does not raise any IndexError** when the index in out of range

In [122]:

```
# slice operator in python list
l = [10,20,30,40,50,60,70,80,90]
l[2:6:2] # from 2nd index to 6-1 index with step 2
```

Out[122]:

```
[30, 50]
```

Slice operator on 1-D numpy array

same rules of python slice operator

Syntax: array_name[begin:end:step]

- **begin,end and step** values can be **either +ve or -ve**. These are optional
- If **begin** is not specified then **0** will be taken **by default**
- If **end** is not specified then **upto end** of the list will be taken
- If **step** is not specified then **1** will be taken **by default**
- **step value cannot be 0**
- If **step value is +ve**, then in **forward direction begin to end-1** will be considered
- If **step value is -ve**, then in **backward direction begin to end+1** will be considered
- In **forward direction**, if **end is 0** then the result will be always **empty**
- In **backward direction**, if **end is -1** then the result will be always **empty**

In [123]:

```
# Slice operator on 1-D array
import numpy as np
a = np.arange(10,101,10)
a
```

Out[123]:

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

In [124]:

```
a[2:5] # from 2nd index to 5-1 index
```

Out[124]:

```
array([30, 40, 50])
```

In [125]:

```
a[::1] # entire array
```

Out[125]:

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

In [126]:

```
a[::-1] # entire array in reverse order
```

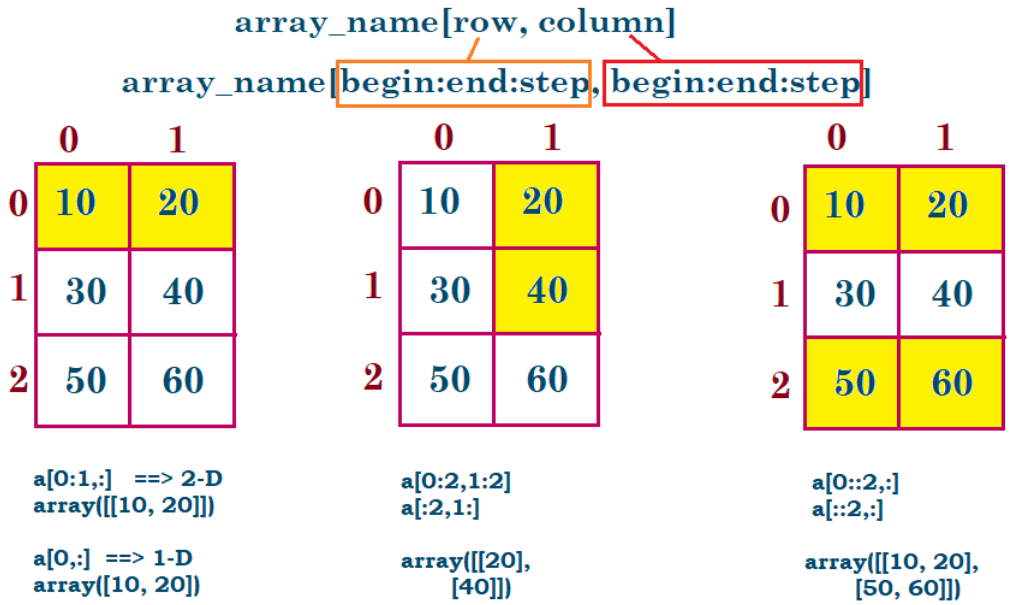
Out[126]:

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

Slice operator on 2-D numpy array

Syntax

- array_name[row,column]
- array_name[begin:end:step,begin:end:step]



In [127]:

```
# slice operator on 2-D array
a = np.array([[10,20],[30,40],[50,60]])
a
```

Out[127]:

```
array([[10, 20],
       [30, 40],
       [50, 60]])
```

Note

- row and colum should be in slice syntax Then only it will return 2-D array
- If any one is index then it will return 1-D array

In [128]:

```
import numpy as np
a = np.array([[10,20],[30,40],[50,60]])
b = a[0,:] # row is in index form, it will return 1-D array
c = a[0:1,:] # it will return 2-D array
print(f"The dimension of b : {b.ndim} and the array b : {b} ")
print(f"The dimension of c : {c.ndim} and the array c : {c} ")
```

```
The dimension of b : 1 and the array b : [10 20]
The dimension of c : 2 and the array c : [[10 20]]
```

In [129]:

```
print(f"a[0:2,1:2] value is : \n {a[0:2,1:2]}")
print(f"a[:2,1:] value is : \n {a[:2,1:]}")
```

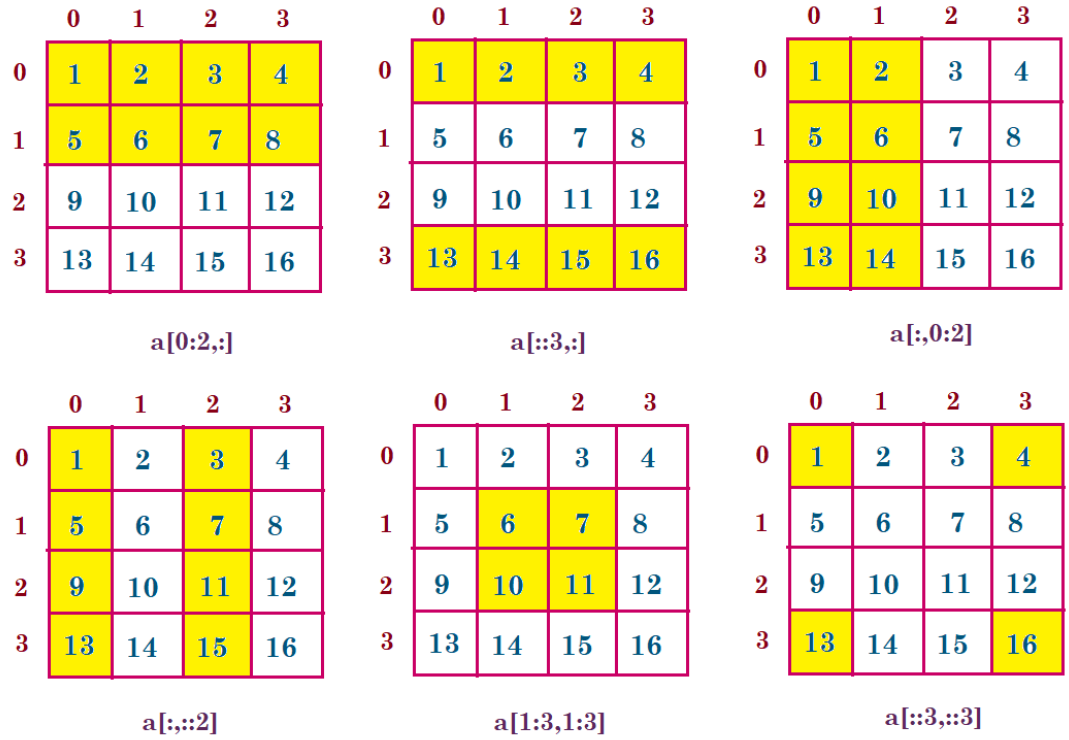
```
a[0:2,1:2] value is :
 [[20]
 [40]]
a[:2,1:] value is :
 [[20]
 [40]]
```

In [130]:

```
print(f"a[0::2,:] value is : \n {a[0::2,:]}")
print(f"a[::2,:] value is : \n {a[::2,:]}")
```

```
a[0::2,:] value is :
 [[10 20]
 [50 60]]
```

```
a[::2,:] value is :
 [[10 20]
 [50 60]]
```



In [131]:

```
# 4 X 4 array
a = np.arange(1,17).reshape(4,4)
a
```

Out[131]:

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

In [132]:

```
print(f"a[0:2,:] value is : \n {a[0:2,:]}")
print(f"a[0::3,:] value is : \n {a[0::3,:]}")
print(f"a[:,:2] value is : \n {a[:,:2]}")
print(f"a[:,::2] value is : \n {a[:,::2]}")
print(f"a[1:3,1:3] value is : \n {a[1:3,1:3]}")
print(f"a[::3,::3] value is : \n {a[::3,::3]}")
```

```
a[0:2,:] value is :
 [[1 2 3 4]
 [5 6 7 8]]
a[0::3,:] value is :
 [[ 1  2  3  4]
 [13 14 15 16]]
a[:,:2] value is :
 [[ 1  2]
 [ 5  6]
 [ 9 10]
 [13 14]]
a[:,::2] value is :
 [[ 1  3]
 [ 5  7]
 [ 9 11]
 [13 15]]
a[1:3,1:3] value is :
 [[ 6  7]
 [10 11]]
a[::3,::3] value is :
 [[ 1  4]
 [13 16]]
```

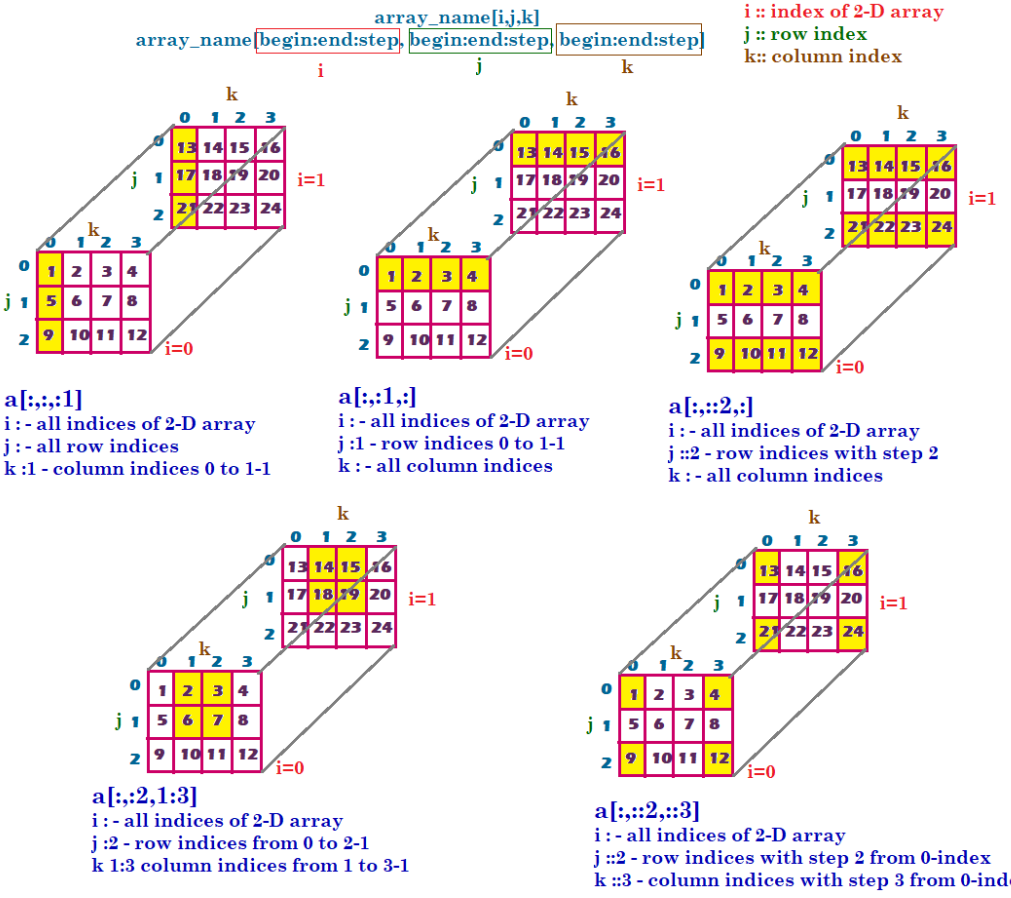
Slice operator on 3-D numpy array

Syntax

array_name[i,j,k]

- i ==> indices of 2-D array (axis-0)
- j ==> row index (axis-1)
- k ==> column index (axis-2)

array_name[begin:end:step,begin:end:step,begin:end:step]



In [133]:

```
# slice operator on 3-D array

import numpy as np
a = np.arange(1,25).reshape(2,3,4)
a
```

Out[133]:

```
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]]])
```

In [134]:

print(f"The slice of a[:,:,:1] :\n {a[:,:,:1]}")

```
The slice of a[:,:,:1] :
 [[[ 1]
  [ 5]
  [ 9]]

 [[13]
  [17]
  [21]]]
```

In [135]:

print(f"The slice of a[:,:1,:] ==> \n {a[:,:1,:]}")

```
The slice of a[:,:1,:] ==>
 [[[ 1  2  3  4]]

 [[13 14 15 16]]]
```

In [136]:

print(f"The slice of a[:,::2,:] ==> \n {a[:,::2,:]}")

```
The slice of a[:,::2,:] ==>
 [[[ 1  2  3  4]
  [ 9 10 11 12]]

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

In [137]:

print(f"The slice of a[:,:2,1:3] ==> \n {a[:,:2,1:3]}")

```
The slice of a[:,:2,1:3] ==>
 [[[ 2  3]
  [ 6  7]]

 [[14 15]
  [18 19]]]
```

In [138]:

print(f"The slice of a[:,::2,::3] ==> \n {a[:,::2,::3]}")

```
The slice of a[:,::2,::3] ==>
 [[[ 1  4]
  [ 9 12]]

 [[13 16]
  [21 24]]]
```

Note:

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

Advanced indexing-- group of elements which are not in order (arbitrary elements)

When the elements are not ordered then we will go for `Advanced indexing`

To access arbitrary elements(elements which are out of order) we should go for Advanced Indexing.

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

- **1-D array** ==> a[i]
- **2-D array** ==> a[i][j]
- **3-D array** ==> a[i][j][k]

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

- **1-D array ==> a[begin:end:step]**
- **2-D array ==> a[begin:end:step,begin:end:step]**
- **3-D array ==> a[begin:end:step,begin:end:step,begin:end:step]**

Accessing multiple arbitrary elements in 1-D array

Syntax :

- **array[x]** ==> **x** can be either **ndarray** or **list**, which represents **required indices**

1st Way(ndarray)

- create an ndarray with required indices in the given original array
- pass that array as argument to the original array

2nd way(list)

- create a list with the required indices in the given original array
- pass that list as argument to the original array

In [139]:

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

Out[139]:

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

1st way(using ndarray)

In [140]:

```
# 1st way (ndarray)
# step1 : create an ndarray with the required indices
# assume that we have to extract 30,50,60,80 elements. Their indices are 2,4,5,8
# create an ndarray with those indices
indices = np.array([2,4,5,8])
indices
```

Out[140]:

```
array([2, 4, 5, 8])
```

In [141]:

```
# step 2: pass the indices as argument to the original array to get the required arbitrary elements
a[indices]
```

Out[141]:

```
array([30, 50, 60, 90])
```

2nd way(using list)

In [142]:

```
# 2nd way (list)
# Step 1: create a list with the required indices of the original array
l = [2,4,5,8]

#Step 2: pass the list as an argument to the original array to get the required arbitrary elements
a[l]
```

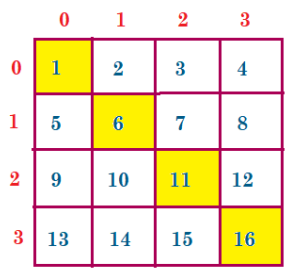
Out[142]:

```
array([30, 50, 60, 90])
```

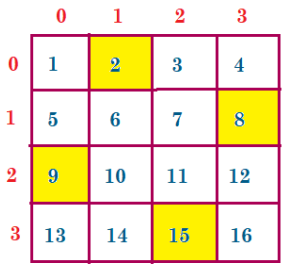
Accessing multiple arbitrary elements in 2-D array

Syntax:

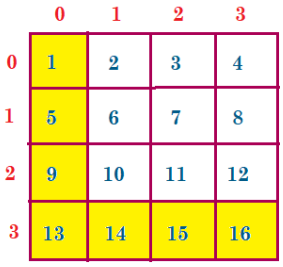
- **a[[row_indices],[column_indices]]**



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



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



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

How access the required elements from the 2-D array

Step 1: find the coordinates of the required elements(row,column)
Assume that required elements are 1,5,9,13,14,15,16
1 :: (0,0) 14 :: (3,1)
5 :: (1,0) 15 :: (3,2)
9 :: (2,0) 16 :: (3,3)
13 :: (3,0)

Step 2 : Arrange them in an order (0,0),(1,0),(2,0),(3,0),(3,1),(3,2),(3,3)

Step 3: create row list with first co-ordinates [0,1,2,3,3,3,3]
create column list with second co-ordinates [0,0,0,0,1,2,3]

Step4: pass these lists as argument to the original array
a[[0,1,2,3,3,3,3,3],[0,0,0,0,1,2,3]]

In [143]:

```
l = [[1,2,3,4],[5,6,7,8],[9,10,11,12],[13,14,15,16]]
a = np.array(l)
a[[0,1,2,3],[0,1,2,3]] #  It select elements from (0,0),(1,1),(2,2) and (3,3)
```

Out[143]:

```
array([ 1,  6, 11, 16])
```

In [144]:

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

Out[144]:

```
array([ 2,  8,  9, 15])
```

In [145]:

 # L-shape
a[[0,1,2,3,3,3,3],[0,0,0,0,1,2,3]]

Out[145]:

```
array([ 1,  5,  9, 13, 14, 15, 16])
```

Note

- In advanced indexing, the required elements will be selected based on indices and with those elements a 1-D array will be created.
- The **result of advanced indexing is always 1-D array only** even though we select 1-D or 2-D or 3-D array as input
- But in **Slicing the dimension of output array is always same as the dimension of input array.**

In [146]:

 # Observations :

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

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-146-c4e1e7026d51> in <module>
      2
      3 # 1. a[[0,1,2],[0,1]]
----> 4 a[[0,1,2],[0,1]]

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

In [147]:

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

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-147-ab01250a8073> in <module>
      1 # 2. a[[0,1],[0,1,2]]
```

```
----> 2 a[[0,1],[0,1,2]]

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

In [148]:

```
# 3. a[[0,1,2],[0]]
a[[0,1,2],[0]]  # it will be taken as (0,0),(1,0),(2,0)
```

Out[148]:

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

In [149]:

```
# 4. a[[0],[0,1,2]]
a[[0],[0,1,2]] # it will be taken as (0,0),(0,1),(0,2)
```

Out[149]:

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

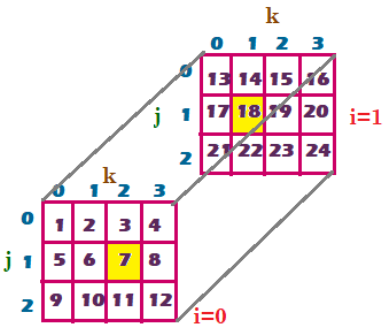
Accessing multiple arbitrary elements in 3-D array

a[i][j][k]

- **i** represents the **index of 2-D array**
- **j** represents **row index**
- **k** represents **column index**

Syntax:

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



Assume that we have to select 7 and 18 elements

Step 1: Find the coordinates of arbitrary elements

7 ::	18::
i : 0	i : 1
j : 1	j : 1
k : 2	k : 1

Step 2: create lists of i,j,k

i : [0,1]
j : [1,1]
k: [2,1]

Step 3: Pass these lists as arguments to the original array

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

In [150]:

```
# accessing the arbitrary elements of 3-D arrays
a = np.arange(1,25).reshape(2,3,4)
a
```

Out[150]:

```
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]]])
```

In [151]:

```
# accesing 7 and 18 from the array
a[[0,1],[1,1],[2,1]]
```

Out[151]:

```
array([ 7, 18])
```

Summary

To access the arbitrary elements of 1-D array

- **a[x] :: x can be either ndarray or list of indices**

To access the arbitrary elements of 2-D array

- **a[[row_indices],[column_indices]]**

To access the arbitrary elements of 3-D array

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

Condition based selection : array_name[condition]

- We can select elements of an array based on condition also

Syntax :: array_name[boolean_array]

- In **boolean_array**, whereever **True** is present the **corresponding value** will be selected

In [152]:

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

Out[152]:

```
array([10, 40])
```

Syntax for condition based selection :: array_name[condition]

- **condition** always return **boolean value**

In [153]:

```
# select elements which are greater than 25 from the given array
a = np.array([10,20,30,40])  # boolean_array should be [False,False,True,True]
# condition will give the boolean value
a>25
```

Out[153]:

```
array([False, False,  True,  True])
```

In [154]:

```
# create boolean array
bool_array = a>25
# pass the bool_array as argument to the original array
a[bool_array]
```

Out[154]:

```
array([30, 40])
```

In [155]:

```
# in single step also we can achieve the functionality
# array_name[condition]
a[a>25]
```

Out[155]:

```
array([30, 40])
```

In [156]:

```
# select then negative numbers from the given array
a = np.array([10,-5,20,40,-3,-1,75])
a[a<0]
```

Out[156]:

```
array([-5, -3, -1])
```

In [157]:

```
## condition based selection is applicatble for 2-D array also
a = np.arange(1,26).reshape(5,5)
a
```

Out[157]:

```
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]])
```

In [158]:

 # select the even numbers from 2-D array
a[a%2==0]

Out[158]:

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

Logical operations

- **logical_and(x1, x2, /[, out, where, ...])** ==> Compute the truth value of **x1 AND x2 element-wise**.
- **logical_or(x1, x2, /[, out, where, casting, ...])** ==> Compute the truth value of **x1 OR x2 element-wise**.
- **logical_not(x, /[, out, where, casting, ...])** ==> Compute the truth value of **NOT x element-wise**.
- **logical_xor(x1, x2, /[, out, where, ...])** ==> Compute the truth value of **x1 XOR x2, element-wise**.

In [159]:

help(np.logical_and)

In [160]:

 # we can select elements by using multiple conditions by using logical operations
 # select the elements which are even and divisible by 5
a = np.arange(1,26).reshape(5,5)
b = np.logical_and(a%2==0,a%5==0)
print(f"Original array a ==> \n {a}")
print(f"Logical array b ==> \n {b}")
print(f"Elements which are even and divisible by 5 ==> \n {a[b]}")

```
Original array a ==>
 [[ 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]]
```

```
Logical array b ==>
 [[False False False False False]
 [False False False False  True]
 [False False False False False]
 [False False False False  True]
 [False False False False False]]
Elements which are even and divisible by 5 ==>
 [10 20]
```

selecting elements based on multiple conditions

We can use `&` for AND condition and `|` for OR condition

- **array_name[(condition_1) & (condition_2)]**
- **array_name[(condition_1) | (condition_2)]**

In [161]:

```
# select the elements which are divisible by 2 and divisible by 3
a = np.arange(1,26).reshape(5,5)
a[(a%2 == 0) & (a%3 ==0)]
```

Out[161]:

```
array([ 6, 12, 18, 24])
```

In [162]:

```
bool_array = np.logical_and(a%2==0,a%5==0)
a[bool_array]
```

Out[162]:

```
array([10, 20])
```

In [163]:

```
# select the elements either even and divisible by 3
a = np.arange(1,26).reshape(5,5)
a[np.logical_or(a%2==0,a%3==0)]
```

Out[163]:

```
array([ 2,  3,  4,  6,  8,  9, 10, 12, 14, 15, 16, 18, 20, 21, 22, 24])
```

In [164]:

```
# select the elments which are divisible by 2 and 3
a = np.arange(1,26).reshape(5,5)
a[(a%2 == 0) & (a%3 ==0)]
```

Out[164]:

```
array([ 6, 12, 18, 24])
```

In [165]:

```
# select the elments which are divisible by either 2 or 3
a = np.arange(1,26).reshape(5,5)
a[(a%2 == 0) | (a%3 ==0)]
```

Out[165]:

```
array([ 2,  3,  4,  6,  8,  9, 10, 12, 14, 15, 16, 18, 20, 21, 22, 24])
```

Python Slicing Vs Numpy Array Slicing Vs Adv. Indexing Vs. Conditional Selection

case-1 : Python slicing

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

In [166]:

```
# Python Slicing  ==> new copy is created
l1 = [10,20,30,40]
l2=l1[::]
l1 is l2
```

Out[166]:

```
False
```

In [167]:

```
# if we made changes in l1 it does not reflect on l2 and vice-versa
l1[0] = 888
print(f"l1::{l1}")
print(f"l2::{l2}")
```

```
l1::[888, 20, 30, 40]
l2::[10, 20, 30, 40]
```

In [168]:

```
l2[1] = 7777
print(f"l1::{l1}")
print(f"l2::{l2}")
```

```
l1::[888, 20, 30, 40]
l2::[10, 7777, 30, 40]
```

case-2 : Numpy Array Slicing:

- A separate copy won't be created and just we are getting **view of the original copy.**
- View is logical entity where as Table is physical entity.(RDBMS)

In [169]:

```
# Numpy array slicing
a = np.arange(10,101,10)
a
```

Out[169]:

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

In [170]:

```
# slicing on numpy array
b = a[0:4] # view is created
print(f"b ==> {b}")
print(f"a ==> {a}")
```

```
b ==> [10 20 30 40]
a ==> [ 10  20  30  40  50  60  70  80  90 100]
```

In [171]:

```
# if we made changes in b it will reflect on a also vice-versa
b[0] = 999
print(f"After modifying the b array value ==> {b}")
print(f"a array value ==> {a}")
```

```
After modifying the b array value ==> [999  20  30  40]
a array value ==> [999  20  30  40  50  60  70  80  90 100]
```

In [172]:

```
a[2] = 888
print(f"After modifying the a array value ==> {a}")
print(f"b array value ==> {b}")
```

```
After modifying the a array value ==> [999  20 888  40  50  60  70  80  90 100]
b array value ==> [999  20 888  40]
```

case-3 : Advanced Indexing and Condition Based Selection

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

In [173]:

```
# Advanced Indexing
a = np.arange(10,101,10)
b = a[[0,2,5]]
print(f" a ==> {a}")
print(f" b ==> {b}")
```

```
 a ==> [ 10  20  30  40  50  60  70  80  90 100]
 b ==> [10 30 60]
```

Slicing Vs Advanced Indexing

Slicing

- The elements should be ordered
- We can't select arbitrary elements
- Conditional based selection is not possible
- Just we will get view but not copy
- Memeory, performance-wise it is the best

Advanced indexing

- The elements need not be ordered
- We can select arbitrary elements
- Conditional based selection is possible
- Just we will get separate copy but not view
- Memeory, performance-wise it is not upto the mark

Summary of syntaxes

Basic Indexing

- 1-D array :: a[i]
- 2-D array :: a[i][j] or a[i,j]
- 3-D array :: a[i][j][k] or a[i,j,k]

Slicing

- 1-D array :: a[begin:end:step]
- 2-D array :: a[begin:end:step,begin:end:step]
- 3-D array :: a[begin:end:step,begin:end:step,begin:end:step]

Advanced Indexing

- 1-D array :: a[x] --> x contains required indices of type ndarray or list
- 2-D array :: a[[row indices],[column indices]]
- 3-D array :: a[[indices of 2D array],[row indices],[column indices]]

Condition based selection

- a[condition] eg: a[a>0]
- This is same for all 1-D, 2-D and 3-D arrays

Chapter-6
How to iterate elements of nd array

Iterate elements of nd array

- Iteration means getting all elements one by one

We can iterate nd arrays in 3 ways

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

By using Python's loops concept

Iterate elemetns of 1-D array

```
In [174]:

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

```
10
20
30
40
50
```

Iterate elemetns of 2-D array

```
In [175]:

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

```
10
20
30
```

```
40
50
60
70
80
90
```

Iterate elemetns of 3-D array

```
In [176]:

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

```
10
20
30
40
50
60
70
80
```

Note: To iterate elements of n-D array, we require n loops.

By using nditer() function

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

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

Iterate elements of 1-D array

```
In [177]:

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

```
10
20
30
40
50
```

Iterate elements of 2-D array

```
In [178]:

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

```
10
20
30
40
50
60
70
80
90
```

Iterate elements of 3-D array

```
In [179]:

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

```
10
20
30
40
50
60
70
80
```

Iterate elements of sliced array also

In [180]:

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

```
10
20
40
50
70
80
```

Using nditer() to get elements of required data type

- We have to use **op_dtypes parameter**

In [181]:

```
import numpy as np
help(np.nditer)
```

```
Help on class nditer in module numpy:

class nditer(builtins.object)
 |  nditer(op, flags=None, op_flags=None, op_dtypes=None, order='K', casting=
'safe', op_axes=None, itershape=None, buffersize=0)
```

In [182]:

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

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-182-64f49c1b6d2f> in <module>
      1 import numpy as np
      2 a = np.array([[10,20,30],[40,50,60],[70,80,90]])
```

```
----> 3 for x in np.nditer(a,op_dtypes=['float']):
      4     print(x)
      5 print(a)

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

In [183]:

```
# 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],[40,50,60],[70,80,90]])
for x in np.nditer(a,flags=['buffered'],op_dtypes=['float']):
  print(x)
print(a)
```

```
10.0
20.0
30.0
40.0
50.0
60.0
70.0
80.0
90.0
[[10 20 30]
 [40 50 60]
 [70 80 90]]
```

In [184]:

```
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
```

```
int32
int32
int32
int32
int32
```

```
int32
int32
int32
```

In [185]:

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

```
int64
int64
int64
int64
int64
int64
int64
int64
```

Normal Python's loops vs nditer()

Python Loops

- n loops are required
- There is no way to specify our required dtype

nditer()

- only one loop is enough
- There is a way to specify our required dtype. For this we have to use `op_dtypes` argument.

By using 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.

Iterate elements of 1-D array:

In [186]:

```
# Iterate elements of 1-D array:
import numpy as np
```

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

```
10 element present at index/position:(0,)
20 element present at index/position:(1,)
30 element present at index/position:(2,)
40 element present at index/position:(3,)
50 element present at index/position:(4,)
60 element present at index/position:(5,)
70 element present at index/position:(6,)
```

Iterate elements of 2-D array:

```
In [187]:

# Iterate elements of 2-D array:
import numpy as np
a = np.array([[10,20,30],[40,50,60],[70,80,90]])
for pos,element in np.ndenumerate(a):
  print(f'{element} element present at index/position:{pos}')
```

```
10 element present at index/position:(0, 0)
20 element present at index/position:(0, 1)
30 element present at index/position:(0, 2)
40 element present at index/position:(1, 0)
50 element present at index/position:(1, 1)
60 element present at index/position:(1, 2)
70 element present at index/position:(2, 0)
80 element present at index/position:(2, 1)
90 element present at index/position:(2, 2)
```

Iterate elements of 3-D array:

```
In [188]:

# Iterate elements of 3-D array:
import numpy as np
a = np.arange(1,25).reshape(2,3,4)
for pos,element in np.ndenumerate(a):
  print(f'{element} element present at index/position:{pos}')
```

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

```
3 element present at index/position:(0, 0, 2)
4 element present at index/position:(0, 0, 3)
5 element present at index/position:(0, 1, 0)
6 element present at index/position:(0, 1, 1)
7 element present at index/position:(0, 1, 2)
8 element present at index/position:(0, 1, 3)
9 element present at index/position:(0, 2, 0)
10 element present at index/position:(0, 2, 1)
11 element present at index/position:(0, 2, 2)
12 element present at index/position:(0, 2, 3)
13 element present at index/position:(1, 0, 0)
14 element present at index/position:(1, 0, 1)
15 element present at index/position:(1, 0, 2)
16 element present at index/position:(1, 0, 3)
17 element present at index/position:(1, 1, 0)
18 element present at index/position:(1, 1, 1)
19 element present at index/position:(1, 1, 2)
20 element present at index/position:(1, 1, 3)
21 element present at index/position:(1, 2, 0)
22 element present at index/position:(1, 2, 1)
23 element present at index/position:(1, 2, 2)
24 element present at index/position:(1, 2, 3)
```

Chapter-7
Arithmetic operators

Arithmetic Operators:

The following are the 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.

In [189]:

```
print(f"10/2 value :: {10/2}")
print(f"10.0/2value :: {10.0/2}")
print(f"10//2 value :: {10//2}")
print(f"10.0//2 value :: {10.0//2}")
```

```
10/2 value :: 5.0
10.0/2value :: 5.0
10//2 value :: 5
10.0//2 value :: 5.0
```

Arithmetic operators for Numpy arrays with scalar:

- scalar means constant numeric value.
- All arithmetic operators are applicable for Numpy arrays with scalar.
- All these operations will be performed at element level.

1-D Array

In [190]:

```
import numpy as np
a = np.array([10,20,30,40])
print(f"a+2 value is :: {a+2}")
print(f"a-2 value is :: {a-2}")
print(f"a*2 value is :: {a*2}")
print(f"a**2 value is :: {a**2}")
print(f"a/2 value is :: {a/2}")
print(f"a//2 value is :: {a//2}")
```

```
a+2 value is :: [12 22 32 42]
a-2 value is :: [ 8 18 28 38]
a*2 value is :: [20 40 60 80]
a**2 value is :: [ 100  400  900 1600]
a/2 value is :: [ 5. 10. 15. 20.]
a//2 value is :: [ 5 10 15 20]
```

2-D Array

In [191]:

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

Out[191]:

```
array([[10, 20, 30],
       [40, 50, 60]])
```

In [192]:

```
print(f"a value is ::\n {a}")
print(f"a+2 value is :: \n {a+2}")
print(f"a-2 value is :: \n {a-2}")
print(f"a*2 value is :: \n {a*2}")
print(f"a**2 value is ::\n {a**2}")
print(f"a/2 value is :: \n {a/2}")
print(f"a//2 value is ::\n {a//2}")
```

```
a value is ::
 [[10 20 30]
 [40 50 60]]


a+2 value is ::
 [[12 22 32]
 [42 52 62]]
a-2 value is ::
 [[ 8 18 28]
 [38 48 58]]
a*2 value is ::
 [[ 20  40  60]
 [ 80 100 120]]
a**2 value is ::
 [[ 100  400  900]
 [1600 2500 3600]]
a/2 value is ::
 [[ 5. 10. 15.]
 [20. 25. 30.]]
a//2 value is ::
 [[ 5 10 15]
 [20 25 30]]
```

ZeroDivisionError

- In **python** Anything by zero including zero/zero also results in : **ZeroDivisionError**
- But in **numpy** there is **no ZeroDivisionError**
- 10/0 ==> Infinity(inf)
- 0/0 ==> undefined(nan--->not a number)

In [193]:

```
# normal Python
print(f"The value of 10/0 :: {10/0}")
print(f"The value of 0/0 :: {0/0}")
```

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
<ipython-input-193-9d387af29aeb> in <module>
      1 # normal Python
----> 2 print(f"The value of 10/0 :: {10/0}")
      3 print(f"The value of 0/0 :: {0/0}")
```

```
ZeroDivisionError: division by zero
```

In [194]:

```
# numpy arrays
a = np.arange(6)
print(f"The value of a/0 :: {a/0}")
```

```
The value of a/0 :: [nan inf inf inf inf inf]

<ipython-input-194-58ff1c7748d1>:3: RuntimeWarning: divide by zero encountere
d in true_divide
  print(f"The value of a/0 :: {a/0}")
<ipython-input-194-58ff1c7748d1>:3: RuntimeWarning: invalid value encountered
in true_divide
  print(f"The value of a/0 :: {a/0}")
```

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

To perform arithmetic operators between numpy arrays, compulsory both arrays should have

- same dimension,
- same shape and
- same size,

otherwise we will get error.

1-D arrays

In [195]:

```
import numpy as np
a = np.array([1,2,3,4])
b = np.array([10,20,30,40])
print(f"Dimension of a : {a.ndim}, size of a :{a.shape} and shape of a : {a.shape}")
print(f"Dimension of b : {b.ndim}, size of a :{b.shape} and shape of a : {b.shape}")
print(f"a array :: {a} and b array :: {b}")
print(f"a+b value is :: {a+b}")
print(f"a-b value is :: {a-b}")
print(f"a*b value is :: {a*b}")
print(f"a**b value is :: {a**b}")
```

```
print(f"a/b value is :: {a/b}")
print(f"a//b value is :: {a//b}")
```

```
Dimension of a : 1, size of a :(4,) and shape of a : (4,)
Dimension of b : 1, size of a :(4,) and shape of a : (4,)
a array :: [1 2 3 4] and b array :: [10 20 30 40]
a+b value is :: [11 22 33 44]
a-b value is :: [ -9 -18 -27 -36]
a*b value is :: [ 10  40  90 160]
a**b value is :: [          1     1048576 -1010140999           0]
a/b value is :: [0.1 0.1 0.1 0.1]
a//b value is :: [0 0 0 0]
```

2-D arrays

```
In [196]:

a = np.array([[1,2],[3,4]])
b = np.array([[5,6],[7,8]])
print(f"Dimension of a : {a.ndim}, size of a :{a.shape} and shape of a : {a.shape}")
print(f"Dimension of b : {b.ndim}, size of a :{b.shape} and shape of a : {b.shape}")
print(f"a array :: \n {a} ")
print(f"b array :: \n {b} ")
print(f"a+b value is :: \n {a+b}")
print(f"a-b value is :: \n {a-b}")
print(f"a*b value is :: \n {a*b}")
print(f"a**b value is :: \n {a**b}")
print(f"a/b value is :: \n {a/b}")
print(f"a//b value is :: \n {a//b}")
```

```
Dimension of a : 2, size of a :(2, 2) and shape of a : (2, 2)
Dimension of b : 2, size of a :(2, 2) and shape of a : (2, 2)
a array ::
 [[1 2]
 [3 4]]
b array ::
 [[5 6]
 [7 8]]
a+b value is ::
 [[ 6  8]
 [10 12]]
a-b value is ::
 [[-4 -4]
```

```
 [-4 -4]]


a*b value is ::
 [[ 5 12]
 [21 32]]
a**b value is ::
 [[    1    64]
 [ 2187 65536]]
a/b value is ::
 [[0.2        0.33333333]
 [0.42857143 0.5       ]]
a//b value is ::
 [[0 0]
 [0 0]]
```

In [197]:

```
a = np.array([10,20,30,40])
b = np.array([10,20,30,40,50])
print(f"Dimension of a : {a.ndim}, size of a :{a.shape} and shape of a : {a.shape}")
print(f"Dimension of b : {b.ndim}, size of a :{b.shape} and shape of a : {b.shape}")
print(f"a+b value is :: \n {a+b}")
```

```
Dimension of a : 1, size of a :(4,) and shape of a : (4,)
Dimension of b : 1, size of a :(5,) and shape of a : (5,)

---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-197-5e5693a9e518> in <module>
      3 print(f"Dimension of a : {a.ndim}, size of a :{a.shape} and shape of
a : {a.shape}")
      4 print(f"Dimension of b : {b.ndim}, size of a :{b.shape} and shape of
a : {b.shape}")
----> 5 print(f"a+b value is :: \n {a+b}")

ValueError: operands could not be broadcast together with shapes (4,) (5,)
```

Equivalent function for arithmetic operators in numpy

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

Note

To use these functions both arrays should be in

- same dimension,
- same size and
- same shape

In [198]:

```
# Using the functions to perform arithmetic operations
import numpy as np
a = np.array([10,20,30,40])
b = np.array([1,2,3,4])
print(f"Dimension of a : {a.ndim}, size of a :{a.shape} and shape of a : {a.shape}")
print(f"Dimension of b : {b.ndim}, size of a :{b.shape} and shape of a : {b.shape}")
print(f"a array :: {a} and b array :: {b}")
print(f"a+b value is :: { np.add(a,b)}")
print(f"a-b value is :: {np.subtract(a,b)}")
print(f"a*b value is :: {np.multiply(a,b)}")
print(f"a/b value is :: {np.divide(a,b)}")
print(f"a//b value is :: {np.floor_divide(a,b)}")
print(f"a%b value is :: {np.mod(a,b)}")
print(f"a**b value is :: {np.power(a,b)}")
```

```
Dimension of a : 1, size of a :(4,) and shape of a : (4,)
Dimension of b : 1, size of a :(4,) and shape of a : (4,)
a array :: [10 20 30 40] and b array :: [1 2 3 4]
a+b value is :: [11 22 33 44]
a-b value is :: [ 9 18 27 36]
a*b value is :: [ 10  40  90 160]
a/b value is :: [10. 10. 10. 10.]
a//b value is :: [10 10 10 10]
a%b value is :: [0 0 0 0]
a**b value is :: [     10     400   27000 2560000]
```

Universal Functions(ufunc)

- The functions which operates element by element on whole array are called Universal functions (ufunc).
- All the above functions are ufunc.
- np.dot() :: Matrix Multiplication/Dot product
- np.multiply() :: Element multiplication

Chapter-8
Broadcasting

Broadcasting

- Generally Arithmetic operations are performed between two arrays are having same dimension, shape and size.
- Eventhough dimensions are different,shapes are different and sizes are different still some arithmetic operations are allowed by Broadcasting
- Broadcasting will be performed automatically by numpy itself and we are not required to perform explicitly.
- Broadcasting won't be possible in all cases.
- Numpy follow some rules to perform Broadcasting. 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: Make sure both arrays should have same dimension

- If the two arrays are of different dimensions, numpy will make equal dimensions. Padded 1's in the shape of lesser dimension array on the left side, until both arrays have same diemension

Eg: Before

- a array has shape :: (4,3) :::: 2-D array
- b array has shape :: (3,) :::: 1-D array

After

- Both arrays a and b are different dimensions.
- By **Rule-1** Numpy will add 1's to the lesser dimension array(here array b). Now the array a is ::: (4,3)

and the array b becomes ::: (1,3) -- By using Rule-1

- Now Both arrays are in same dimension array a:: 2-D and array-b :: 2-D
  a array has shape :: (4,3) :::: 2-D array
  b array has shape :: (1,3) :::: 2-D array

Rule-2:

- If the size of two arrays does not match in any dimension, then the arrays with size equal to 1 in that dimension will be increased to size of other dimension to match

Eg:

- From Rule-1 we got a ==> (4,3) and b ==> (1,3)
- Here first co-ordinate of a => 4 and b => 1. Sizes are different
- According to Rule-2 the array with size 1 (here arrray b with size 1) will be increased to 4 (corresponding size of array a).
- Second co-ordinate of a => 3 and b => 3. Sizes are matched.
- After applying Rule-2 the dimensions of a and b are changed as follows

  a array has shape ==> (4,3) ====> 2-D array

  b array has shape ==> (4,3) ====> 2-D array

- Now both the arrays are having same dimension, shape and size. So we can perform any arithmetic operations

Note

- In any dimension, the sizes are not matched and neither equal to 1, then we will get error, Numpy does not able to perform broadcasting between those arrays
- The data will be reused from the same input array.
- If the rows are required then reuse existing rows.
- If columns are required then reuse existing columns.
- The result is always higher dimension of input arrays.

Eg: Broadcasting between (3,2,2) and (3,) possible or not

Before applying Rule-1

- a :: (3,2,2) - 3-D array
- b :: (3,) - 1-D array
- Here bothe are different dimensions. By applying the Rule-1 the Numpy changes the array b as (1,1,3)

After applying Rule-1

- a :: (3,2,2) - 3-D array
- b :: (1,1,3) - 3-D array
- Now both arrays are in same dimension

Before applying Rule-2

- a :: (3,2,2) - 3-D array
- b :: (1,1,3) - 3-D array
- By applying Rule-2 Numpy changes the array b to (3,2,3) because it has size 1's. It will be replaced with corresponding sizes of the arrray a

After applying Rule-2

- a :: (3,2,2) - 3-D array
- b :: (3,2,3) - 3-D array
- Now here array a and b are having same dimensions, but different shapes. So Numpy unable to perform broadcasting.

In [199]:

```
# Broadcasting between 1-D arrays of different sizes
import numpy as np
a = np.array([10,20,30,40])
b = np.array([1,2,3])
# a : (4,) b: (3,).
# Both are having same diemensions. But sizes are different so arithmetic operation is not performed
# Broadcasting by Numpy will be failed while performing arithmetic operation in this case
a+b
```

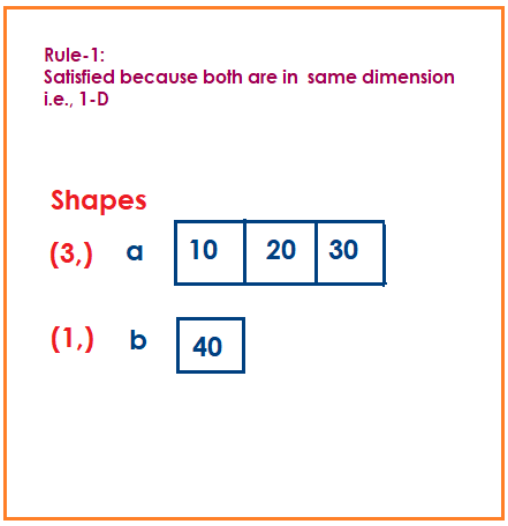
```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-199-e084bfeb9eb8> in <module>
      6 # Both are having same diemensions. But sizes are different so arithm
```

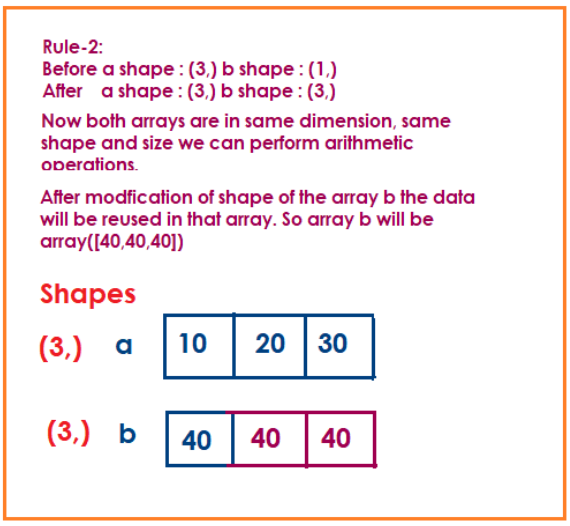
```
etic operation is not performed
      7 # Broadcasting by Numpy will be failed while performing arithmetic op
eration in this case
----> 8 a+b

ValueError: operands could not be broadcast together with shapes (4,) (3,)
```

In [200]:

```
# Broadcasting between 1-D arrays
a = np.array([10,20,30])
b = np.array([40])
print(f"Shape of a : {a.shape}")
print(f"Shape of b : {b.shape}")
print(f"a+b : {a+b}")
```

```
Shape of a : (3,)
Shape of b : (1,)
a+b : [50 60 70]
```



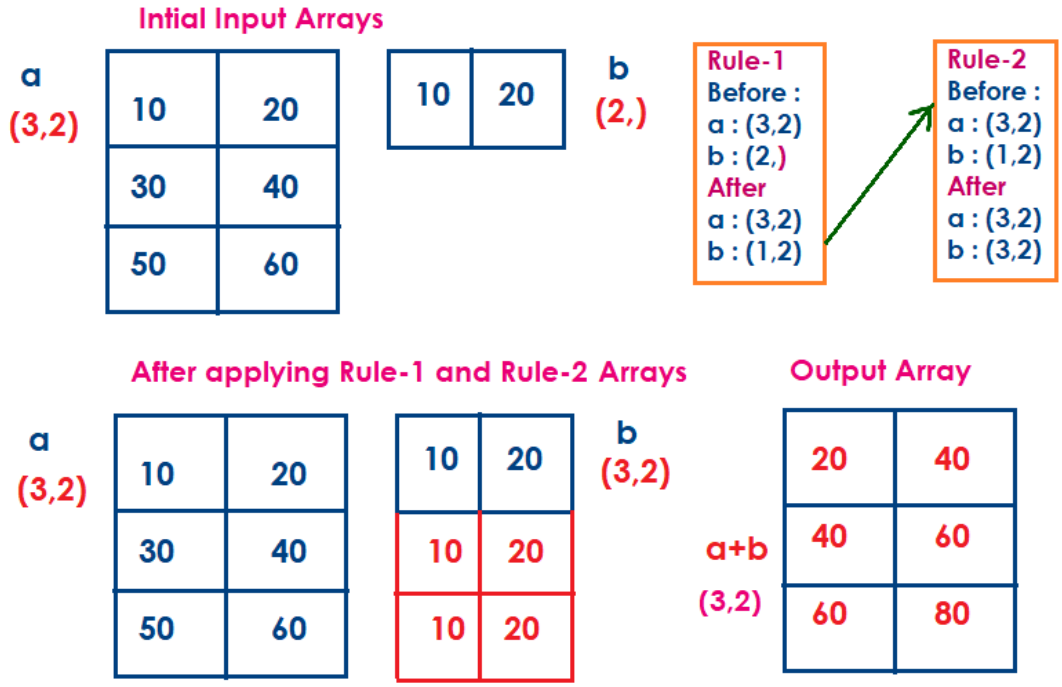


In [201]:

```
# Broadcasting between 2-D and 1-D arrays
a = np.array([[10,20],[30,40],[50,60]])
b = np.array([10,20])
print(f"Shape of the array a : {a.shape}")
print(f"Shape of the array b : {b.shape}")
print(f"array a :\n {a}")
print(f"array b :\n {b}")
print("Arithmetic operations :")
```

```
print(f"Addition operation a+b :\n {a+b}")
print(f"Subtraction operation a-b :\n {a-b}")
print(f"Multiplication operation a*b:\n {a*b}")
print(f"Division operation a/b:\n {a/b}")
print(f"Floor division operation a//b:\n {a//b}")
print(f"Modulo operation a%b:\n {a%b}")
```

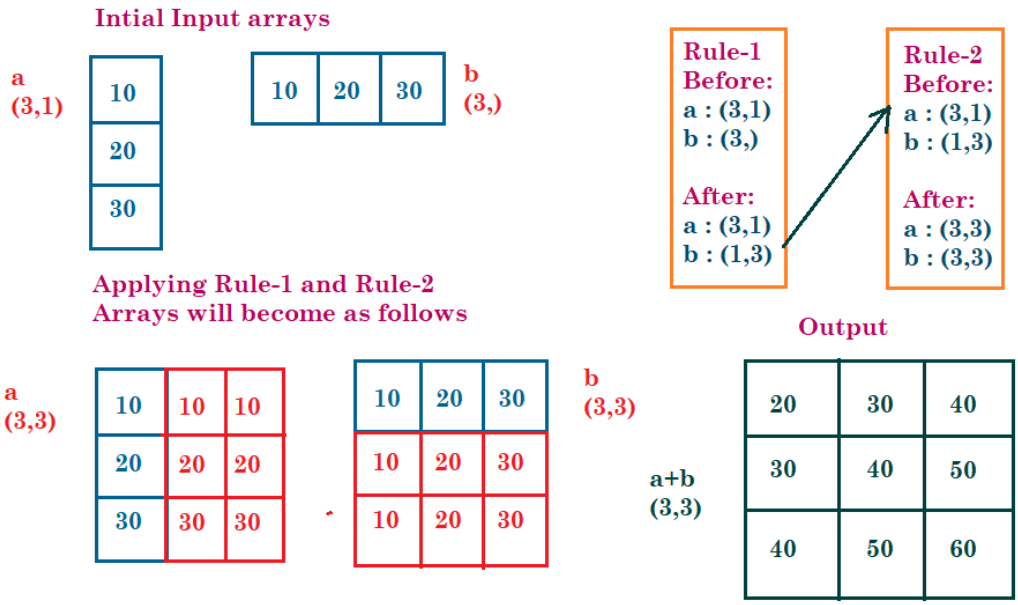
```
Shape of the array a : (3, 2)
Shape of the array b : (2,)
array a :
 [[10 20]
 [30 40]
 [50 60]]
array b :
 [10 20]
Arithmetic operations :
Addition operation a+b :
 [[20 40]
 [40 60]
 [60 80]]
Subtraction operation a-b :
 [[ 0  0]
 [20 20]
 [40 40]]
Multiplication operation a*b:
 [[ 100  400]
 [ 300  800]
 [ 500 1200]]
Division operation a/b:
 [[1. 1.]
 [3. 2.]
 [5. 3.]]
Floor division operation a//b:
 [[1 1]
 [3 2]
 [5 3]]


Modulo operation a%b:
 [[0 0]
 [0 0]
 [0 0]]
```




```
In [202]:

# Broadcasting between 1-D array and 2-D array
a = np.array([[10],[20],[30]])
b = np.array([10,20,30])
print(f"Shape of the array a : {a.shape}")
print(f"Shape of the array b : {b.shape}")
print(f"array a :\n {a}")
print(f"array b :\n {b}")
print("Arithmetic operations :")
print(f"Addition operation a+b :\n {a+b}")
print(f"Subtraction operation a-b :\n {a-b}")
print(f"Multiplication operation a*b:\n {a*b}")
print(f"Division operation a/b:\n {a/b}")
print(f"Floor division operation a//b:\n {a//b}")
print(f"Modulo operation a%b:\n {a%b}")
```

```
Shape of the array a : (3, 1)
Shape of the array b : (3,)
array a :
 [[10]
 [20]
 [30]]
```

```
array b :
 [10 20 30]
Arithmetic operations :
Addition operation a+b :
 [[20 30 40]
 [30 40 50]
 [40 50 60]]
Subtraction operation a-b :
 [[  0 -10 -20]
 [ 10   0 -10]
 [ 20  10   0]]
Multiplication operation a*b:
 [[100 200 300]
 [200 400 600]
 [300 600 900]]
Division operation a/b:
 [[1.         0.5        0.33333333]
 [2.         1.         0.66666667]
 [3.         1.5        1.        ]]
Floor division operation a//b:
 [[1 0 0]
 [2 1 0]
 [3 1 1]]
Modulo operation a%b:
 [[ 0 10 10]
 [ 0  0 20]
 [ 0 10  0]]
```


Intial Input arrays
a
(3,1)
10
20
30
10 20 30
b
(3,)
Rule-1
Before:
a : (3,1)
b : (3,)
After:
a : (3,1)
b : (1,3)
Rule-2
Before:
a : (3,1)
b : (1,3)
After:
a : (3,3)
b : (3,3)
Applying Rule-1 and Rule-2
Arrays will become as follows
a
(3,3)
10 10 10
20 20 20
30 30 30
-
10 20 30
10 20 30
10 20 30
b
(3,3)
Output
a+b
(3,3)
20 30 40
30 40 50
40 50 60

Chapter-9
Array Manipulation functions

Array Manipulation functions

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.
- The Data should not be changed. Input size and output size shoule be matched, other-wise we will get the ValueError
- No chnage in the data. So new object won't be created. We will get just **View.** If we perform any change in the original array, that change will be reflected to reshaped array and vice-versa
- We can specify unknown dimension size as -1, but only once.

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

- **C style** ==> **Row major order(default)** : it will consider 1st row, 2nd row and so on
- **Fortran Style(F)** ==> **Column major order** : it will considier 1st column, 2nd column and so on

We can use either reshape() function of the numpy array or reshape() method of ndarray

- np.reshape(a,new_shape) ==> Functional style
- ndarray_obj.reshape(new_shape) ==> Object oriented style

In [203]:

```
# one shape to any other shape

# (10,) --> (5,2),(2,5),(10,1),(1,10) ==> any no. of views
# (24,) --> (3,8), (6,4),(2,3,4), (2,2,2,4) all are valid
```

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)

by using np.reshape() ==> Functional Style

In [204]:

```
# While using reshape() function, make sure the sizes should be matched otherwise we will get error.
import numpy as np
a = np.arange(1,11)
print(f"array : {a}")
b = np.reshape(a,(5,3))
```

```
array : [ 1  2  3  4  5  6  7  8  9 10]

---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-204-2a87cd9be4ff> in <module>
      3 a = np.arange(1,11)
      4 print(f"array : {a}")
----> 5 b = np.reshape(a,(5,3))

<__array_function__ internals> in reshape(*args, **kwargs)

F:\Users\Gopi\anaconda3\lib\site-packages\numpy\core\fromnumeric.py in reshap
e(a, newshape, order)
```

```
    297             [5, 6]])
    298     """
--> 299     return _wrapfunc(a, 'reshape', newshape, order=order)
    300
    301


F:\Users\Gopi\anaconda3\lib\site-packages\numpy\core\fromnumeric.py in _wrapf
unc(obj, method, *args, **kwds)
     56
     57     try:
---> 58         return bound(*args, **kwds)
     59     except TypeError:
     60         # A TypeError occurs if the object does have such a method in
its

ValueError: cannot reshape array of size 10 into shape (5,3)
```

In [205]:

```
# converting 1-D array to 2-D array
a = np.arange(1,11)
print(f"array : {a}")
b = np.reshape(a,(5,2))
print(f"Converting 1-D(a) array to 2-D(b) array : \n {b} ")
```

```
array : [ 1  2  3  4  5  6  7  8  9 10]
Converting 1-D(a) array to 2-D(b) array :
 [[ 1  2]
 [ 3  4]
 [ 5  6]
 [ 7  8]
 [ 9 10]]
```

In [206]:

```
# View is created but not copy Any change in the original array will be reflected to
the reshaped
# array
a = np.arange(1,16)
b = np.reshape(a,(5,3))
print("*"*80)
print(f"Original array : {a}")
print(f"Reshaped array :\n {b}")
```

```
print("*"*80)
print("After modification in the orginal array")
a[0] = 111
print(f"Original array : {a}")
print(f"Reshaped array :\n {b}")
print("*"*80)
print("After modification in the reshaped array")
b[2,1] = 888
print(f"Reshaped array :\n {b}")
print(f"Original array : {a}")
print("*"*80)
```

```
********************************************************************************
Original array : [ 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15]
Reshaped array :
 [[ 1  2  3]
 [ 4  5  6]
 [ 7  8  9]
 [10 11 12]
 [13 14 15]]
********************************************************************************
After modification in the orginal array
Original array : [111   2   3   4   5   6   7   8   9  10  11  12  13  14  15
]
Reshaped array :
 [[111   2   3]
 [  4   5   6]
 [  7   8   9]
 [ 10  11  12]
 [ 13  14  15]]
********************************************************************************
After modification in the reshaped array
Reshaped array :
 [[111   2   3]
 [  4   5   6]
 [  7 888   9]
 [ 10  11  12]
 [ 13  14  15]]
Original array : [111   2   3   4   5   6   7 888   9  10  11  12  13  14  15]
********************************************************************************
```

by using ndarray_obj.reshape() ==> Object oriented style

In [207]:

```
a = np.arange(1,25)
b = a.reshape((2,3,4))
print("*"*80)
print(f"Original array : \n {a}")
print(f"Reshaped array :\n {b}")
print("*"*80)
```

```
********************************************************************************
Original 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]
Reshaped 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]]]
********************************************************************************
```

order='C'

- **C style :: Row major order**
- It will consider 1st row, 2nd row and so on

In [208]:

```
# C style order :: Row major order
a = np.arange(12)
b = a.reshape((3,4))
print("*"*80)
print(f"Original array : {a}")
print(f"Reshaped array :\n {b}")
print("*"*80)
```

```
********************************************************************************
Original array : [ 0  1  2  3  4  5  6  7  8  9 10 11]
Reshaped array :
 [[ 0  1  2  3]
 [ 4  5  6  7]
```

```
 [ 8  9 10 11]]
**************************************************************************
```

order='F'

- **Fortran style :: Column major order**
- It will consider 1st column, 2nd column and so on

In [209]:

```
# Fortran style order :: Column major order
a = np.arange(12)
b = a.reshape((3,4),order='F')
print("*"*80)
print(f"Original array : {a}")
print(f"Reshaped array :\n {b}")
print("*"*80)
```

```
**************************************************************************
Original array : [ 0  1  2  3  4  5  6  7  8  9 10 11]
Reshaped array :
 [[ 0  3  6  9]
 [ 1  4  7 10]
 [ 2  5  8 11]]
**************************************************************************
```

unknown dimension with -1

- We can use unknown dimension with -1, only once.
- Numpy will decide and replace -1 with the required dimension
- we can use any -ve number instead of -1
- 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**

In [210]:

```
a = np.arange(1,11)
b = a.reshape((5,-1))
print(f"Shape of array b :: {b.shape}")
```

```
Shape of array b :: (5, 2)
```

In [211]:

```
a = np.arange(1,11)
b = a.reshape((-1,5))
print(f"Shape of array b :: {b.shape}")
```

```
Shape of array b :: (2, 5)
```

In [212]:

```
a = np.arange(1,11)
b = a.reshape((-1,-1))
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-212-dbdbcca6ff23> in <module>
      1 a = np.arange(1,11)
----> 2 b = a.reshape((-1,-1))

ValueError: can only specify one unknown dimension
```

Conclusions of reshape() function

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

resize()

Syntax:

numpy.resize(a, new_shape)

- output array: can be any dimension,any shape,any size
- input size and output size need not to be matched
- The data may be changed
- We will get the copy but the view
- We can get the new data by np.resize() and a.resize()
- unknown dimension -1 is not applicable in this resize()

by using np.resize() ==> Functional Style

Syntax: numpy.resize(a, new_shape)

- If new_size requires more elements repeat elements of the input array
- new object will be created

In [213]:

```
a = np.arange(1,6)
b = np.resize(a,(2,4))
print(f"Original array : {a}")
print(f"Reshaped array :\n {b}")
```

```
Original array : [1 2 3 4 5]
Reshaped array :
 [[1 2 3 4]
 [5 1 2 3]]
```

In [214]:

```
c = np.resize(a,(2,2))
print(f" array c : \n {c}")
```

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

In [215]:

```
# original array a is not modified when we call np.resize() function
a
```

Out[215]:

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

by using ndarray_obj.resize() ==> Object oriented style

Syntax : ndarray_object.resize(new_shape, refcheck=True)

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

In [216]:

```
m = np.arange(1,6)
print(f"Original array :\n {m}")
m.resize((4,2))
print(f"After calling resize() method in ndarray :\n {m}")
```

```
Original array :
 [1 2 3 4 5]
After calling resize() method in ndarray :
 [[1 2]
 [3 4]
 [5 0]
 [0 0]]
```

In [217]:

```
# original array get modified when we call ndarray_obj.resize() method
m
```

Out[217]:

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

numpy.resize()	ndarray.resize()
It is library function present in numpy module	It is method present in ndarray class
It will creates a new array and returns it.	It won't return new array and existing array will be modified(inline modification).
If the new_shape requires more elements then repeated copies of original array will be used.	If the new_shape requires more elements then extra elements filled with zeros

reshape()	resize()
It is to change shape of array, but not size.	It is to change size of the array, automatically shape and data may be changed.
It won't create new array object and just we will get view of existing array.	It will create new array object with required new shape.
If we perform any changes in the reshaped copy, automatically those changes will be reflected in original copy and vice-versa	If we perform any changes in the resized array, those changes won't be reflected in original copy.
The reshape will be happend without changing original data.	There may be a chance of data change in resize() either expansion or shriking
The sizes must be matched	The sizes need not be matched.
In unknown dimension we can use -1.	-1, such type of story not applicable.

flatten() method

- 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.

  C-style====>row major order

  F-stype===>column major order

- It is method present in ndarray class but not numpy library function.

  a.flatten()--->valid

  np.flatten()-->invalid

- It will create a new array and returns it (ie copy but not view). If any change is made in the original array it won't be reflected in flatten copy and vice-versa
- **The output of flatten method is always 1D array**

In [218]:

```
import numpy as np
help(np.flatten)
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-218-c04b96076733> in <module>
      1 import numpy as np
----> 2 help(np.flatten)

F:\Users\Gopi\anaconda3\lib\site-packages\numpy\__init__.py in __getattr__(at
tr)
    301                 return Tester
    302
--> 303             raise AttributeError("module {!r} has no attribute "
    304                                  "{!r}".format(__name__, attr))
    305

AttributeError: module 'numpy' has no attribute 'flatten'
```

In [219]:

```
help(np.ndarray.flatten)
```

```
Help on method_descriptor:

flatten(...)
    a.flatten(order='C')

    Return a copy of the array collapsed into one dimension.

    Parameters
    ----------
    order : {'C', 'F', 'A', 'K'}, optional
        'C' means to flatten in row-major (C-style) order.
        'F' means to flatten in column-major (Fortran-
        style) order. 'A' means to flatten in column-major
        order if `a` is Fortran *contiguous* in memory,
        row-major order otherwise. 'K' means to flatten
        `a` in the order the elements occur in memory.
        The default is 'C'.
```

In [220]:

```
# converting 2-D array to 1-D array ==> C style
a = np.arange(6).reshape(3,2)
b = a.flatten()
print(f"Original array :\n {a}")
print(f"Flatten array :\n {b}")
```

```
Original array :
 [[0 1]
 [2 3]
 [4 5]]
Flatten array :
 [0 1 2 3 4 5]
```

In [221]:

```
# change the value of Original array. The changes wont be reflected in Flatten array
a[0,0] = 8888
print(f"Original array :\n {a}")
print(f"Flatten array :\n {b}")
```

```
Original array :
 [[8888    1]
 [   2    3]
 [   4    5]]
Flatten array :
 [0 1 2 3 4 5]
```

In [222]:

```
# converting the 2-D array to 1-D array ==> Fortran style F
a = np.arange(6).reshape(3,2)
b = a.flatten('F')
print(f"Original array :\n {a}")
print(f"Flatten array :\n {b}")
```

```
Original array :
 [[0 1]
 [2 3]
 [4 5]]
Flatten array :
 [0 2 4 1 3 5]
```

In [223]:

```
# Converting 3-D to 1-D ==> C Style
a = np.arange(1,19).reshape(3,3,2)
b = a.flatten()
print(f"Original array :\n {a}")
print(f"Flatten array :\n {b}")
```

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

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

In [224]:

```
# Converting 3-D to 1-D ==> Fortran Style(F)
a = np.arange(1,19).reshape(3,3,2)
b = a.flatten('F')
print(f"Original array :\n {a}")
print(f"Flatten array :\n {b}")
```

```
Original array :
 [[[ 1  2]
  [ 3  4]
  [ 5  6]]

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

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

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

flat variable

- It is a 1-D iterator over the array.
- This is a 'numpy.flatiter' instance.
- **ndarray.flat** ==> Return a flat iterator over an array.
- **ndarray.flatten** ==> Returns a flattened copy of an array.

In [225]:

```
# help on flatiter
help(np.flatiter)
```

In [226]:

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

Out[226]:

```
<numpy.flatiter at 0x1aac9a8e770>
```

In [227]:

```
a.flat[4]
```

Out[227]:

```
5
```

In [228]:

```
# iterate over flat
for x in a.flat: print(x)
```

```
1
2
3
4
```

```
5
6
```

ravel() ==> it is a library function in numpy module and a method in ndarray class

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

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

  np.ravel()-->valid
- a.ravel(order='C')
  C-style====>row major order

  F-stype===>column major order
- It returns **view but not copy**. If we made change in the Original array then the changes will be reflected in ravel copy also vice-versa

Note: The output of ravel() method is always 1D array

In [229]:

```
# Library function help
help(np.ravel)
```

```
Help on function ravel in module numpy:

ravel(a, order='C')
    Return a contiguous flattened array.

    A 1-D array, containing the elements of the input, is returned.

    A copy is made only if needed.
```

In [230]:

```
# ndarray class method help
help(np.ndarray.ravel)
```

```
Help on method_descriptor:

ravel(...)
    a.ravel([order])

    Return a flattened array.

    Refer to `numpy.ravel` for full documentation.

    See Also
    --------
    numpy.ravel : equivalent function

    ndarray.flat : a flat iterator on the array.
```

In [231]:

```
# using ravel() method in ndarray class
a = np.arange(24).reshape(2,3,4)
b = a.ravel()
print(f"Original array :\n {a}")
print(f"Ravel array :\n {b}")
```

```
Original 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]]]
Ravel 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]
```

In [232]:

```
b[0] = 8888
print(f"Ravel array :\n {b}")
print(f"Original array :\n {a}")
```

```
Ravel array :
 [8888    1    2    3    4    5    6    7    8    9   10   11   12   13
   14   15   16   17   18   19   20   21   22   23]
Original array :
```

```
[[[8888    1    2    3]
  [   4    5    6    7]
  [   8    9   10   11]]

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

In [233]:

```
# using ravel() function in numpy module
a = np.arange(18).reshape(6,3)
b = np.ravel(a)
print(f"Original array :\n {a}")
print(f"Ravel array :\n {b}")
```

```
Original array :
 [[ 0  1  2]
 [ 3  4  5]
 [ 6  7  8]
 [ 9 10 11]
 [12 13 14]
 [15 16 17]]
Ravel array :
 [ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17]
```

flatten()	ravel()
It can used to flatten n-D array to 1-D array and a separate 1-D array object will be created.	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.
If we perform any changes in the flatten() copy, then those changes won't be reflected in the original copy.	If we perform any changes in the ravel() copy, then those changes will be reflected in the original copy.
flatten() method operates slower than ravel() as it is required to create a new array object.	ravel() method operates faster than flatten() as it is not required to create a new array object and it just returns a view.
flatten() is not numpy library level function and it is a method present in ndarray class. ndarrayObj.flatten() --->valid numpy.flatten(a) --->invalid	ravel() is both numpy library level function and ndarray class method. ndarrayObj.ravel() --->valid numpy.ravel(a) --->valid

transpose() => both numpy library function and ndarray method

- 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: shapes in the dimension

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

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \qquad A^T = \begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix}$$

In [234]:

```
import numpy as np
a = np.array([[1,2],[3,4]])
atrans = np.transpose(a)
print(f"Original Array : \n {a}")
print(f"Transposed Array : \n {atrans}")
```

```
Original Array :
 [[1 2]
 [3 4]]
```

```
Transposed Array :
 [[1 3]
 [2 4]]
```

transpose() :: numpy library function

In [235]:

```
import numpy as np
help(np.transpose)
```

```
Help on function transpose in module numpy:

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

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

without specifying the 'axes' parameter while calling the transpose() function

When we call the transpose() function without specifying the axes parameter then input dimension will be simply reversed

- 1-D array (4,) ==> Transposed Array ::(4,) No change
- 2-D array (2,3) ==> Trasposed Array :: (3,2)
- 3-D array (2,3,4) ==> Transposed Array :: (4,3,2)
- 4-D array (2,2,3,4)==> Transposed Array :: (4,3,2,2)

If we are not specifying the axes parameter

dimensions will be reversed

Original Array Shape	Transpose Array Shape
1-D :: (4,)	(4,) No change
2-D :: (2,3)	(3,2)
3-D :: (2,3,4)	(4,3,2)
4-D :: (2,2,3,4)	(4,3,2,2)

In [236]:

```
# Tranpose of 1-D array ==> no impact
a  = np.array([10,20,30,40])
atrans = np.transpose(a)
print(f"Original Array : \n {a}")
print(f"Transposed Array : \n {atrans}")
print(f"Original Array shape :  {a.shape}")
print(f"Transposed Array shape: {atrans.shape}")
```

```
Original Array :
 [10 20 30 40]
Transposed Array :
 [10 20 30 40]
Original Array shape :  (4,)
Transposed Array shape: (4,)
```

In [237]:

```
# Transpose of 2-D
# In 2-D array, because of transpose, rows will become columns and columns will become rows.
a  = np.arange(1,7).reshape(2,3)
atrans = np.transpose(a)
```

```
print(f"Original Array : \n {a}")
print(f"Transposed Array : \n {atrans}")
print(f"Original Array shape :  {a.shape}")
print(f"Transposed Array shape: {atrans.shape}")
```

```
Original Array :
 [[1 2 3]
 [4 5 6]]
Transposed Array :
 [[1 4]
 [2 5]
 [3 6]]
Original Array shape :  (2, 3)
Transposed Array shape: (3, 2)
```

3-D array shape : (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 then the shape will become 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

In [238]:

```
# Transpose of 4-D
a  = np.arange(1,49).reshape(2,2,3,4)
atrans = np.transpose(a)
print(f"Original Array : \n {a}")
print(f"Transposed Array : \n {atrans}")
print(f"Original Array shape :  {a.shape}")
print(f"Transposed Array shape: {atrans.shape}")
```

```
Original 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]]]]
Transposed Array :
 [[[[ 1 25]
   [13 37]]

  [[ 5 29]
   [17 41]]

  [[ 9 33]
   [21 45]]]


 [[[ 2 26]
   [14 38]]

  [[ 6 30]
   [18 42]]

  [[10 34]
   [22 46]]]


 [[[ 3 27]
   [15 39]]

  [[ 7 31]
```

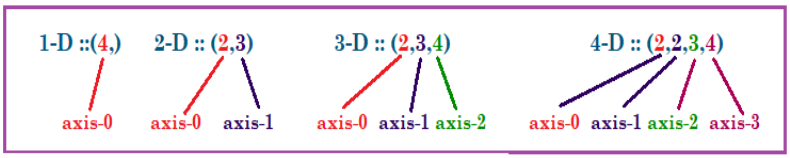
```
   [19 43]]

  [[11 35]
   [23 47]]]


 [[[ 4 28]
   [16 40]]

  [[ 8 32]
   [20 44]]

  [[12 36]
   [24 48]]]]
Original Array shape :  (2, 2, 3, 4)
Transposed Array shape: (4, 3, 2, 2)
```

by specifying the 'axes' parameter while calling the transpose() function

- axes parameter describes in which order we have to take axes.
- It is very helpful for 3-D and 4-D arrays.
- We can specify the customized dimensions

transpose(a, axes=None) axes values can provided in list or tuple form

If we are specifying the axes parameter
customized dimensions can be possible



Original Array Shape	Transposed Array Shape
1-D :: (4,)	(4,)
2-D :: (2,3)	(3,2) –>axes=(1,0) or axes=(0,1)
3-D :: (2,3,4)	(2,4,3), (3,2,4), (3,4,2), (4,3,2),(4,2,3) (2,4,3) –> axes=(0,2,1) (3,4,2) –> axes=(1,2,0) (3,2,4) –> axes=(1,0,2) (4,3,2) –> axes=(2,1,0) (4,2,3) –> axes=(2,0,1)

In [239]:

```
# 1-D array transpose() with axes parameter
a  = np.array([10,20,30,40])
atrans = np.transpose(a,axes=0)
print(f"Original Array : \n {a}")
print(f"Transposed Array : \n {atrans}")
print(f"Original Array shape :  {a.shape}")
print(f"Transposed Array shape: {atrans.shape}")
```

```
Original Array :
 [10 20 30 40]
Transposed Array :
 [10 20 30 40]
```

```
Original Array shape :  (4,)
Transposed Array shape: (4,)
```

In [240]:

```
# 2-D array transpose() with axes parameter
a  = np.arange(1,7).reshape(2,3)
atrans = np.transpose(a,axes=(0,1))
print(f"Original Array : \n {a}")
print(f"Transposed Array : \n {atrans}")
print(f"Original Array shape :  {a.shape}")
print(f"Transposed Array shape: {atrans.shape}")
```

```
Original Array :
 [[1 2 3]
 [4 5 6]]
Transposed Array :
 [[1 2 3]
 [4 5 6]]
Original Array shape :  (2, 3)
Transposed Array shape: (2, 3)
```

In [241]:

```
# 2-D array transpose() with axes parameter
a  = np.arange(1,7).reshape(2,3)
atrans = np.transpose(a,axes=(1,0))
print(f"Original Array : \n {a}")
print(f"Transposed Array : \n {atrans}")
print(f"Original Array shape :  {a.shape}")
print(f"Transposed Array shape: {atrans.shape}")
```

```
Original Array :
 [[1 2 3]
 [4 5 6]]
Transposed Array :
 [[1 4]
 [2 5]
 [3 6]]
Original Array shape :  (2, 3)
Transposed Array shape: (3, 2)
```

In [242]:

```
# 3-D array transpose() with axes parameter
# (2,3,4) ==> Original Array shape
# 2--->2-D arrays (axis-0 : 0)
# 3--->3 rows in every 2-D array (axis-1 : 1)
# 4--->4 columns in every 2-D array (axis-2 : 2)
# (2,4,3) ==> Customized Transposed array shape

a  = np.arange(1,25).reshape(2,3,4)
atrans = np.transpose(a,axes=(0,2,1))
print(f"Original Array : \n {a}")
print(f"Transposed Array : \n {atrans}")
print(f"Original Array shape :  {a.shape}")
print(f"Transposed Array shape: {atrans.shape}")
```

```
Original 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]]]
Transposed Array :
 [[[ 1  5  9]
  [ 2  6 10]
  [ 3  7 11]
  [ 4  8 12]]

 [[13 17 21]
  [14 18 22]
  [15 19 23]
  [16 20 24]]]
Original Array shape :  (2, 3, 4)
Transposed Array shape: (2, 4, 3)
```

In [243]:

```
# 3-D array transpose() with axes parameter
# (2,3,4) ==> Original Array shape
# 2--->2-D arrays (axis-0 : 0)
# 3--->3 rows in every 2-D array (axis-1 : 1)
```

```
# 4--->4 columns in every 2-D array (axis-2 : 2)
# (3,2,4) ==> Customized Transposed array shape

a  = np.arange(1,25).reshape(2,3,4)
atrans = np.transpose(a,axes=(1,0,2))
print(f"Original Array : \n {a}")
print(f"Transposed Array : \n {atrans}")
print(f"Original Array shape :  {a.shape}")
print(f"Transposed Array shape: {atrans.shape}")
```

```
Original 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]]]
Transposed Array :
 [[[ 1  2  3  4]
  [13 14 15 16]]

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

 [[ 9 10 11 12]
  [21 22 23 24]]]
Original Array shape :  (2, 3, 4)
Transposed Array shape: (3, 2, 4)
```

In [244]:

```
### Note: If we repeat the same axis multiple times then we will get error.
b = np.transpose(a,axes=(2,2,1))
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-244-31651c2bfc31> in <module>
      1 ### Note: If we repeat the same axis multiple times then we will get
error.
----> 2 b = np.transpose(a,axes=(2,2,1))

<__array_function__ internals> in transpose(*args, **kwargs)
```

```
F:\Users\Gopi\anaconda3\lib\site-packages\numpy\core\fromnumeric.py in transp
ose(a, axes)
    656
    657     """
--> 658     return _wrapfunc(a, 'transpose', axes)
    659
    660


F:\Users\Gopi\anaconda3\lib\site-packages\numpy\core\fromnumeric.py in _wrapf
unc(obj, method, *args, **kwds)
     56
     57     try:
---> 58         return bound(*args, **kwds)
     59     except TypeError:
     60         # A TypeError occurs if the object does have such a method in
its

ValueError: repeated axis in transpose
```

transpose() :: ndarray class method

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

In [245]:

```
import numpy as np
help(np.ndarray.transpose)
```

In [246]:

```
# 3-D Array
a  = np.arange(1,25).reshape(2,3,4)
atrans = a.transpose((0,2,1))
print(f"Original Array : \n {a}")
print(f"Transposed Array : \n {atrans}")
print(f"Original Array shape :  {a.shape}")
print(f"Transposed Array shape: {atrans.shape}")
```

```
Original 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]]]
Transposed Array :
 [[[ 1  5  9]
  [ 2  6 10]
  [ 3  7 11]
  [ 4  8 12]]

 [[13 17 21]
  [14 18 22]
  [15 19 23]
  [16 20 24]]]
Original Array shape :  (2, 3, 4)
Transposed Array shape: (2, 4, 3)
```

T :: variable in ndarray class

- We can use shortcut representation of the transpose with T variable
- ndarray_obj.T ==> it will simply reverse the dimensions
- Customized dimensions are possible with T variable

In [247]:

```
# By using T variable
a  = np.arange(1,25).reshape(2,3,4)
atrans = a.T
print(f"Original Array : \n {a}")
print(f"Transposed Array : \n {atrans}")
print(f"Original Array shape :  {a.shape}")
print(f"Transposed Array shape: {atrans.shape}")
```

```
Original 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]]]
Transposed 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]]]
Original Array shape :  (2, 3, 4)
Transposed Array shape: (4, 3, 2)
```

reshape Vs 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.**

eg1:

- **input: (3,4)** ==> output: (4,3)

eg2:

- **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)
- **transpose is special case of reshape**

Various possible syntaxes for transpose():

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

Note: 1,3,5 lines are equal wrt functionality.

swapaxes() ==> numpy library function and ndarray class method

- 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.

swapaxes() :: numpy library function

In [248]:

```
import numpy as np
help(np.swapaxes)
```

```
Help on function swapaxes in module numpy:

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

In [249]:

```
a = np.arange(1,7).reshape(3,2)
aswap = np.swapaxes(a,0,1)
print(f"Original Array : \n {a}")
print(f"Swapped Array : \n {atrans}")
print(f"Original Array shape :  {a.shape}")
print(f"Swapped Array shape: {aswap.shape}")
```

```
Original Array :
 [[1 2]
 [3 4]
 [5 6]]
Swapped 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]]]
Original Array shape :  (3, 2)
Swapped Array shape: (2, 3)
```

In [250]:

```
# 2-D array there is no difference when we swap
a = np.arange(1,7).reshape(3,2)
b = np.swapaxes(a,0,1)
c = np.swapaxes(a,1,0)
print(f"Swapped Array b : \n {b}")
print(f"Swapped Array c : \n {c}")
print(f"Swapped Array b shape :  {b.shape}")
print(f"Swapped Array c shape :  {c.shape}")
```

```
Swapped Array b :
 [[1 3 5]
 [2 4 6]]
Swapped Array c :
 [[1 3 5]
 [2 4 6]]
Swapped Array b shape :  (2, 3)
Swapped Array c shape :  (2, 3)
```

In [251]:

```
# 3-D array
a = np.arange(1,25).reshape(2,3,4)
aswap = np.swapaxes(a,0,2)  # 0 and 2 axes values will be swapped : (4,3,2)
print(f"Original Array : \n {a}")
print(f"Swapped Array : \n {atrans}")
print(f"Original Array shape :  {a.shape}")
print(f"Swapped Array shape: {aswap.shape}")
```

```
Original 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]]]
Swapped 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]]]
Original Array shape :  (2, 3, 4)
Swapped Array shape: (4, 3, 2)
```

In [252]:

```
# 3-D array

a = np.arange(1,25).reshape(2,3,4)
aswap = np.swapaxes(a,0,1)  # 0 and 1 axes values will be swapped : (3,2,4)
print(f"Original Array : \n {a}")
print(f"Swapped Array : \n {atrans}")
print(f"Original Array shape :  {a.shape}")
print(f"Swapped Array shape: {aswap.shape}")
```

```
Original 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]]]
```

```
Swapped 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]]]
Original Array shape :  (2, 3, 4)
Swapped Array shape: (3, 2, 4)
```

swapaxes() :: ndarray class method

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

In [253]:

```
import numpy as np
help(np.ndarray.swapaxes)
```

```
Help on method_descriptor:

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

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

    Refer to `numpy.swapaxes` for full documentation.

    See Also
    --------
    numpy.swapaxes : equivalent function
```

In [254]:

```
# 3-D array
a = np.arange(1,25).reshape(2,3,4)
aswap = a.swapaxes(0,1)  # 0 and 1 axes values will be swapped : (3,2,4)
print(f"Original Array : \n {a}")
print(f"Swapped Array : \n {atrans}")
print(f"Original Array shape :  {a.shape}")
print(f"Swapped Array shape: {aswap.shape}")
```

```
Original 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]]]
Swapped 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]]]
Original Array shape :  (2, 3, 4)
Swapped Array shape: (3, 2, 4)
```

transpose() Vs swapaxes():

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

Chapter-10
Joining of multiple arrays into a single array

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()**

concatenate()

In [255]:

```
import numpy as np
help(np.concatenate)
```

```
Help on function concatenate in module numpy:

concatenate(...)
    concatenate((a1, a2, ...), axis=0, out=None, dtype=None, casting="same_ki
nd")

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

Syntax

concatenate(...)

- 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 along an existing axis

axis ==> based on which axis we have to perform concatination

- axis=0(default) :: vertical concatination will happens
- axis=1 :: Horizontal concatination will happens
- axis=None :: First the arrays will be flatten(converted to 1-D array) and then concatination will be performed on the resultant arrays

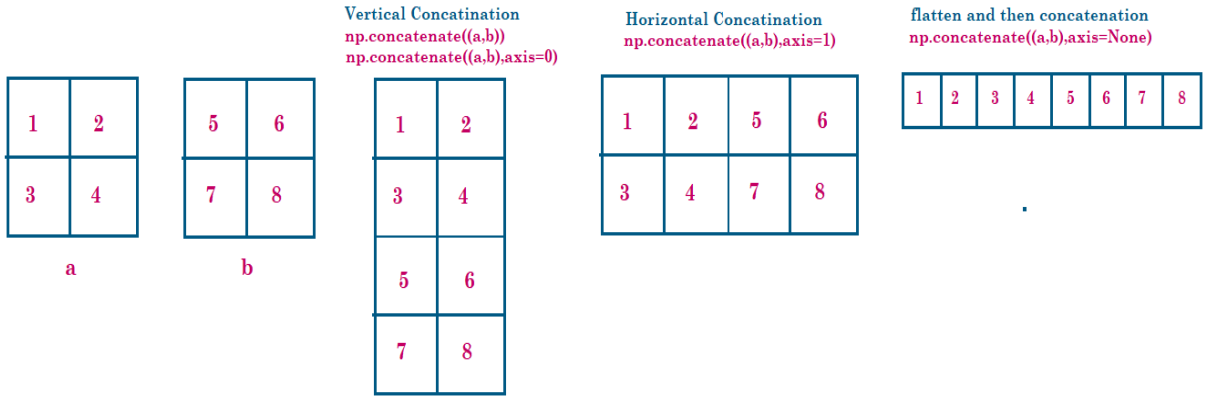
out ==> destination array, where we have to store concatenation result.

In [256]:

```
# 1-D array
import numpy as np
a = np.arange(4)
b = np.arange(5)
c = np.arange(3)
np.concatenate((a,b,c))
```

Out[256]:

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



In [257]:

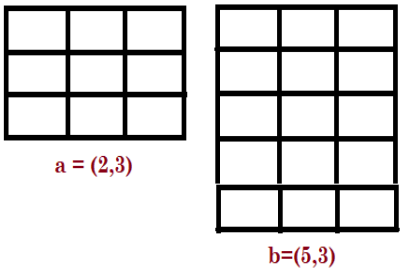
```
# 2-D array
import numpy as np
a = np.array([[1,2],[3,4]])
b = np.array([[5,6],[7,8]])
# concatination by providing axis paraameter

# Vertical Concatenation
vcon = np.concatenate((a,b))
```

```
vcon1 = np.concatenate((a,b),axis=0)

# Horizontal Concateation
hcon = np.concatenate((a,b),axis=1)

# flatten and then concatenation
flatt = np.concatenate((a,b),axis=None)

print(f"array a ==> \n {a}")
print(f"array b ==> \n {b}")
print(f"Without specifying axis parameter ==> \n {vcon}")
print(f"Specifying axis=0 a ==> \n {vcon1}")
print(f"Specifying axis=1 ==> \n {hcon}")
print(f"Specifying axis=None ==> \n {flatt}")
```

```
array a ==>
 [[1 2]
 [3 4]]
array b ==>
 [[5 6]
 [7 8]]
Without specifying axis parameter ==>
 [[1 2]
 [3 4]
 [5 6]
 [7 8]]
Specifying axis=0 a ==>
 [[1 2]
 [3 4]
 [5 6]
 [7 8]]
Specifying axis=1 ==>
 [[1 2 5 6]
 [3 4 7 8]]
Specifying axis=None ==>
 [1 2 3 4 5 6 7 8]
```

Rules

- We can join any number of arrays, but all arrays should be of same dimension.
- The sizes of all axes, except concatenation axis should be same.
- The shapes of resultant array and out array must be same.

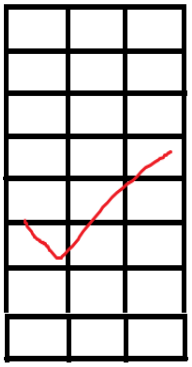


axis=0
size in a :: 2
size in b :: 5

axis=1
size in a :: 3
size in b :: 3

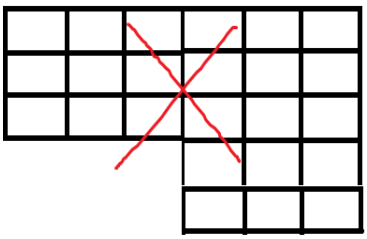
np.concatenate((a,b),axis=0)

except axis=0 consider the sizes of axis=1 in both arrays. They are same(3,3)
So Vertical Concatenation is possible



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

except axis=1 consider the sizes of axis=0 in both arrays. They are different(2,5).
So Horizontal Concatenation is not possible



In [258]:

```
# Rule-2 demonstration
import numpy as np

a = np.arange(6).reshape(2,3)
b = np.arange(15).reshape(5,3)
print(f"array a ==> \n {a}")
print(f"array b ==> \n {b}")

# axis=0 ==> Vertical concatenation
vcon = np.concatenate((a,b),axis=0)
print(f"Vertical Concatenation array ==> \n{vcon}")

# axis=1 ==> Horizontal Concatenation
hcon = np.concatenate((a,b),axis=1)
print(f"Horizontal Concatenation array ==> \n{hcon}")
```

```
array a ==>
 [[0 1 2]
 [3 4 5]]
array b ==>
 [[ 0  1  2]
 [ 3  4  5]
 [ 6  7  8]
 [ 9 10 11]
 [12 13 14]]
```

```
Vertical Concatenation array ==>
[[ 0  1  2]
 [ 3  4  5]
 [ 0  1  2]
 [ 3  4  5]
 [ 6  7  8]
 [ 9 10 11]
 [12 13 14]]

---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-258-e115434e606d> in <module>
     12
     13 # axis=1 ==> Horizontal Concatenation
---> 14 hcon = np.concatenate((a,b),axis=1)
     15 print(f"Horizontal Concatenation array ==> \n{hcon}")

<__array_function__ internals> in concatenate(*args, **kwargs)

ValueError: all the input array dimensions for the concatenation axis must ma
tch exactly, but along dimension 0, the array at index 0 has size 2 and the a
rray at index 1 has size 5
```

Storing the result using 'out' parameter

- concatenate(...)
- concatenate((a1, a2, ...), axis=0, out=None, dtype=None, casting="same_kind")
- Join a sequence of arrays along an `existing axis`.
- we can store result in an array after concatenation using 'out' parameter, but the result and out must be in same shape

In [259]:

```
# example for out parameter
import numpy as np
a = np.arange(4)
b = np.arange(5)
c = np.empty(9) # default dtype for empty is float
np.concatenate((a,b),out=c)
```

Out[259]:

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

In [260]:

```
c
```

Out[260]:

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

In [261]:

```
# if the shape of result and out differs then we will get error : ValueError
import numpy as np
a = np.arange(4)
b = np.arange(5)
c = np.empty(10) # default dtype is float
np.concatenate((a,b),out=c)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-261-580400b85648> in <module>
      4 b = np.arange(5)
      5 c = np.empty(10) # default dtype is float
----> 6 np.concatenate((a,b),out=c)

<__array_function__ internals> in concatenate(*args, **kwargs)

ValueError: Output array is the wrong shape
```

using 'dtype' parameter

- we can specifyt the required data type using `dtype` parameter

In [262]:

```
# Demo for dtype parameter
import numpy as np
a = np.arange(4)
b = np.arange(5)
np.concatenate((a,b),dtype=str)
```

Out[262]:

```
array(['0', '1', '2', '3', '0', '1', '2', '3', '4'], dtype='<U11')
```

Note:

- We can use either `out` or `dtype` .
- We cannot use both `out` and `dtype` simultaneously because `out` has its own data type

In [263]:

```
# Demo for both out dtype parameter
import numpy as np
a = np.arange(4)
b = np.arange(5)
c = np.empty(9,dtype=str)
np.concatenate((a,b),out=c,dtype=str)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-263-67ff4979a9b2> in <module>
      4 b = np.arange(5)
      5 c = np.empty(9,dtype=str)
----> 6 np.concatenate((a,b),out=c,dtype=str)

<__array_function__ internals> in concatenate(*args, **kwargs)

TypeError: concatenate() only takes `out` or `dtype` as an argument, but both
were provided.
```

Concatenation of 1-D arrays

- we can concatenate any number of 1-D arrays at a time
- For 1-D arrays there exists only one axis i.e., axis-0

In [264]:

```
# Demo for concatenation of three 1-D arrays
a = np.arange(4)
b = np.arange(5)
c = np.arange(3)
np.concatenate((a,b,c),dtype=int)
```

Out[264]:

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

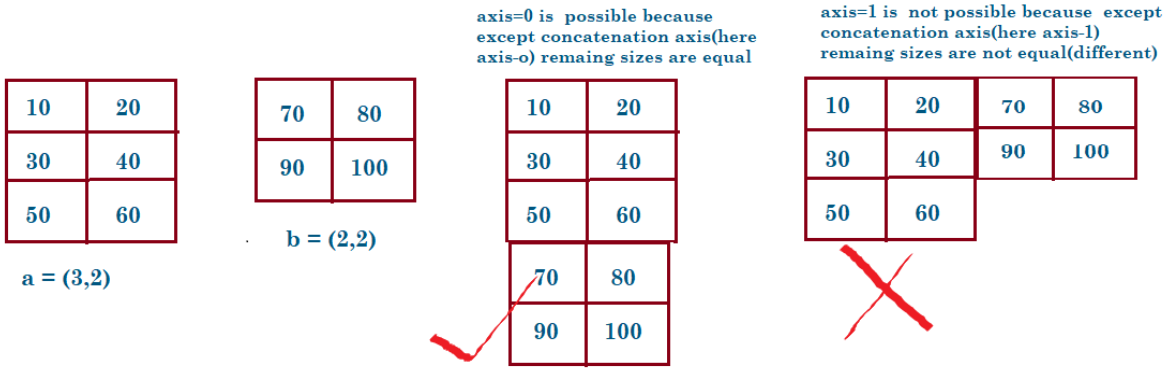
Concatenation of 2-D arrays

- we can concatenate any number of 2-D arrays at a time
- For 2-D arrays there exists two axes i.e., axis-0 and axis-1

axis-0 ==> represents number of rows

axis-1 ==> represents number of columns

- we can perform concatenation either axis-0 or axis-1
- size of all dimensions(axes) must be matched except concatenation axis.



In [265]:

```
# Demo for concatenation of two 2-D arrays
import numpy as np

a = np.array([[10,20],[30,40],[50,60]])
b = np.array([[70,80],[90,100]])
print(f"a array :: \n {a}")
print(f"b array :: \n {b}")
print(f"a shape : {a.shape}")
print(f"b shape : {b.shape}")

# concatenation on axis=0 ==> Vertical concatenation
vcon = np.concatenate((a,b),axis=0)
print(f"Concatenation based on axis-0(vertical) :: \n {vcon}")
```

```
# concatenation on axis=0 ==> Horizontal concatenation
hcon = np.concatenate((a,b),axis=1)
print(f"Concatenation based on axis-1(vertical) :: \n {hcon}")
```

```
a array ::
 [[10 20]
 [30 40]
 [50 60]]
b array ::
 [[ 70  80]
 [ 90 100]]
a shape : (3, 2)
b shape : (2, 2)
Concatenation based on axis-0(vertical) ::
 [[ 10  20]
 [ 30  40]
 [ 50  60]
 [ 70  80]
 [ 90 100]]

---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-265-cd0ec72a4bb8> in <module>
     14
     15 # concatenation on axis=0 ==> Horizontal concatenation
---> 16 hcon = np.concatenate((a,b),axis=1)
     17 print(f"Concatenation based on axis-1(vertical) :: \n {hcon}")

<__array_function__ internals> in concatenate(*args, **kwargs)

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

Concatenation of 3-D arrays

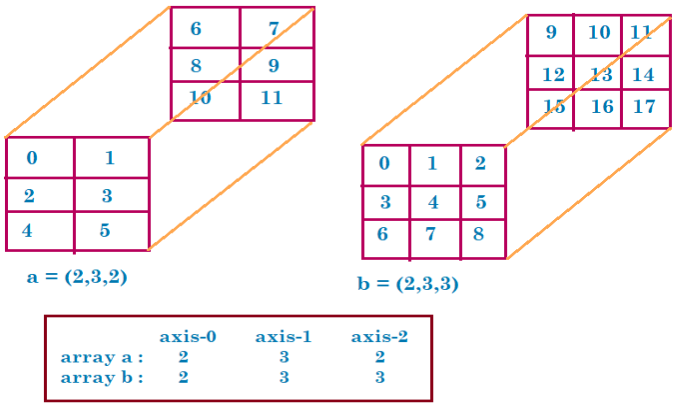
- we can concatenate any number of 3-D arrays at a time
- For 3-D arrays there exists three axes i.e., axis-0,axis-1 and axis-2

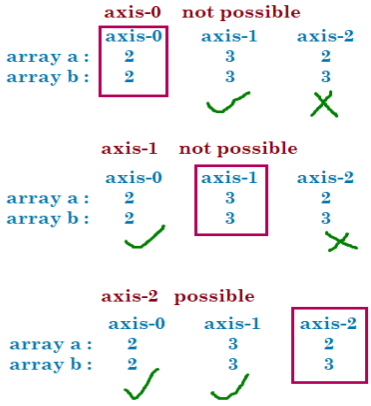
axis-0 ==> represents number of 2-D arrays

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

axis-2 ==> represents number of columns in every 2-D array

- we can perform concatenation on axis-0, axis-1, axis-2 (existing axis)
- size of all dimensions(axes) must be matched except concatenation axis.





3-D array Concatenation

In [266]:

```
# Demo for concatenation of 3-D arrays
import numpy as np
a = np.arange(12).reshape(2,3,2)
b = np.arange(18).reshape(2,3,3)
print(f"array a : \n {a}")
print(f"array b : \n {b}")

# concatenation along axis=0 ==> not possible in this case
np.concatenate((a,b),axis=0)
```

```
array a :
 [[[ 0  1]
  [ 2  3]
  [ 4  5]]

 [[ 6  7]
  [ 8  9]
  [10 11]]]
array b :
 [[[ 0  1  2]
  [ 3  4  5]
```

```
  [ 6  7  8]]

 [[ 9 10 11]
  [12 13 14]
  [15 16 17]]]

---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-266-ae4dcfa205e6> in <module>
      7
      8 # concatenation along axis=0 ==> not possible in this case
----> 9 np.concatenate((a,b),axis=0)

<__array_function__ internals> in concatenate(*args, **kwargs)

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

In [267]:

```
# concatenation along axis=1 ==> not possible in this case
np.concatenate((a,b),axis=1)
```

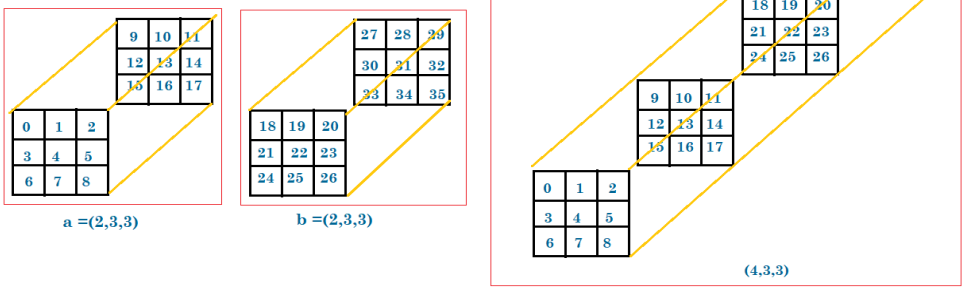
```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-267-a94538b0b908> in <module>
      1 # concatenation along axis=1 ==> not possible in this case
----> 2 np.concatenate((a,b),axis=1)

<__array_function__ internals> in concatenate(*args, **kwargs)

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

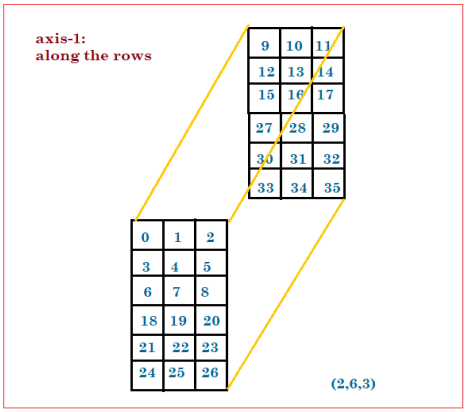
In [268]:

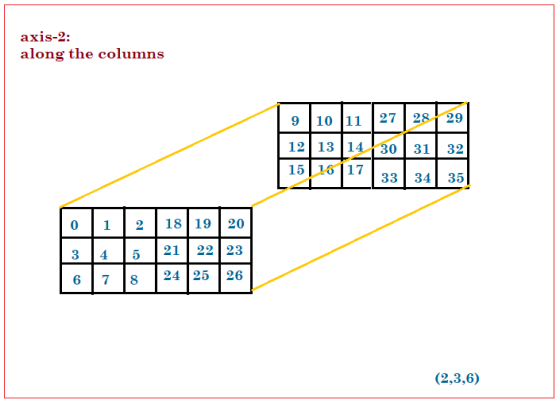
```
# concatenation along axis=2 ==> possible in this case
np.concatenate((a,b),axis=2)
```

Out[268]:

```
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]]])
```







In [269]:

```
# Demo for concatenation of 3-D array in all axes
import numpy as np

a = np.arange(18).reshape(2,3,3)
b = np.arange(18,36).reshape(2,3,3)
print(f"array a : \n {a}")
print(f"array b : \n {b}")
print(f"array a shape:  {a.shape}")
print(f"array b shape:  {b.shape}")
```

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

 [[ 9 10 11]
  [12 13 14]
  [15 16 17]]]
array b :
 [[[18 19 20]
  [21 22 23]
  [24 25 26]]

 [[27 28 29]
  [30 31 32]
  [33 34 35]]]
array a shape:  (2, 3, 3)
array b shape:  (2, 3, 3)
```

In [270]:

```
# concatenation along axis-0
axis0_result = np.concatenate((a,b),axis=0)
print(f"Concatenation along axis-0 : \n {axis0_result}")
print(f"Shape of the resultant array : {axis0_result.shape}")
```

```
Concatenation along axis-0 :
 [[[ 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]]]
Shape of the resultant array : (4, 3, 3)
```

In [271]:

```
# concatenation along axis-1
axis1_result = np.concatenate((a,b),axis=1)
print(f"Concatenation along axis-0 : \n {axis1_result}")
print(f"Shape of the resultant array : {axis1_result.shape}")
```

```
Concatenation along axis-0 :
 [[[ 0  1  2]
  [ 3  4  5]
  [ 6  7  8]
  [18 19 20]
  [21 22 23]
  [24 25 26]]

 [[ 9 10 11]
  [12 13 14]
  [15 16 17]
  [27 28 29]
  [30 31 32]
  [33 34 35]]]
Shape of the resultant array : (2, 6, 3)
```

In [272]:

```
# concatenation along axis-2
axis2_result = np.concatenate((a,b),axis=2)
print(f"Concatenation along axis-0 : \n {axis2_result}")
print(f"Shape of the resultant array : {axis2_result.shape}")
```

```
Concatenation along axis-0 :
 [[[ 0  1  2 18 19 20]
  [ 3  4  5 21 22 23]
  [ 6  7  8 24 25 26]]

 [[ 9 10 11 27 28 29]
  [12 13 14 30 31 32]
  [15 16 17 33 34 35]]]
Shape of the resultant array : (2, 3, 6)
```

Summary

If we concatenate any n-D array the result will be same as the input array dimension

- 1-D + 1-D = 1-D
- 2-D + 2-D = 2-D
- 3-D + 3-D = 3-D

Concatenation is always based on existing axis

All input arrays must be in same dimension

For 2-D arrays after concatenation

- for axis-0 ==> number of 2-D arrays are increased and rows and columns remains unchanged
- for axis-1 ==> Number of rows will be increased and the 2-D arrays and columns unchanged
- for axis-2 ==> Number of columns will be increased and the 2-D arrays and rows unchanged

Note:

- 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.

stack()

- All input arrays must have same shape
- The resultant stacked array has one more dimension than the input arrays.
- The joining is always based on new axis of the `newly created array`

1-D + 1-D = 2-D

2-D + 2-D = 3-D

In [273]:

```
# help on stack
import numpy as np
help(np.stack)
```

```
Help on function stack in module numpy:

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.
```

Stacking 1-D array

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

In [274]:

```
import numpy as np

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

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-274-4046b86fb736> in <module>
      3 a = np.array([10,20,30])
      4 b = np.array([40,50,60,70])
----> 5 np.stack((a,b))

<__array_function__ internals> in stack(*args, **kwargs)

F:\Users\Gopi\anaconda3\lib\site-packages\numpy\core\shape_base.py in stack(a
rrays, axis, out)
    425     shapes = {arr.shape for arr in arrays}
```

```
    426     if len(shapes) != 1:
--> 427         raise ValueError('all input arrays must have the same shape')
    428
    429     result_ndim = arrays[0].ndim + 1

ValueError: all input arrays must have the same shape
```

Stacking between 1-D arrays

- The resultant array will be one more dimension i.e., 2-D array
- newly created array is 2-D and it has two axes ==> axis-0 and axis-1
- so we can perform stacking on axis-0 and axis-1

Stacking along axis=0 in 1-D array

- axis-0 means stack elements of input array row wise
- Read row wise from input arrays and arrange row wise in result array.

In [275]:

```
# stacking using axis=0
a = np.array([10,20,30])
b = np.array([40,50,60])
resultant_array = np.stack((a,b))  # default axis=0
print(f"Resultant array : \n {resultant_array}")
print(f"Resultant array shape: {resultant_array.shape}")
```

```
Resultant array :
 [[10 20 30]
 [40 50 60]]
Resultant array shape: (2, 3)
```

Stacking along axis=1 in 1-D array

- axis-1 means stack elements of input array column wise
- Read row wise from input arrays and arrange column wise in result array.

In [276]:

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

```
print(f"Resultant array : \n {resultant_array}")
print(f"Resultant array shape: {resultant_array.shape}")
```

```
Resultant array :
 [[10 40]
 [20 50]
 [30 60]]
Resultant array shape: (3, 2)
```

Stacking 2-D array

The resultant array will be: 3-D array

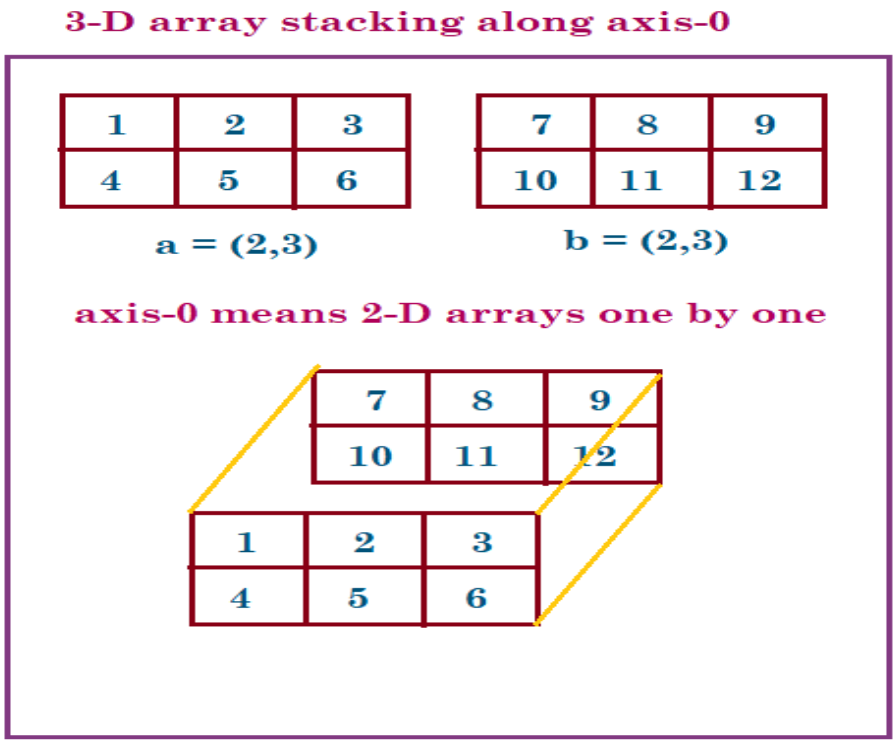
3-D array shape:(x,y,z)

- x-->axis-0 ---->The number of 2-D arrays
- y-->axis-1 ---->The number of rows in every 2-D array
- z-->axis-2 ----> The number of columns in every 2-D array

axis-0 means 2-D arrays one by one

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

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



In [277]:

```
# stacking of 2-D arrays ==> axis=0
# axis-0 means 2-D arrays one by one:
import numpy as np
a = np.array([[1,2,3],[4,5,6]])
b = np.array([[7,8,9],[10,11,12]])
np.stack((a,b)) # by defulat np.stack((a,b),axis=0)
```

Out[277]:

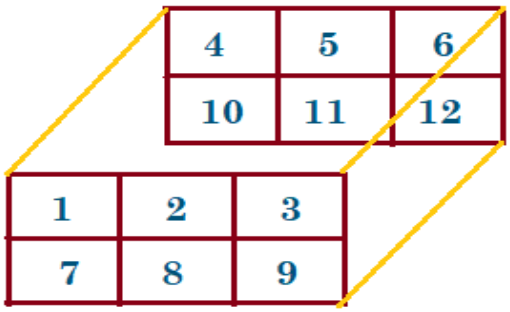
```
array([[[ 1,  2,  3],
        [ 4,  5,  6]],

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

3-D array stacking along axis-1

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

1. Take first row from first array a(1,2,3) make it first row of the resultant array
2. Take first row from second array b(7,8,9) make it second row of the resultant array.
3. combine these rows to form a 2-D array
4. Repeat the steps1,2,3 for remaing rows of the arrays

first 2-D array	second 2-D array
[[1 2 3], [7 8 9]]	[[4,5,6], [10,11,12]]



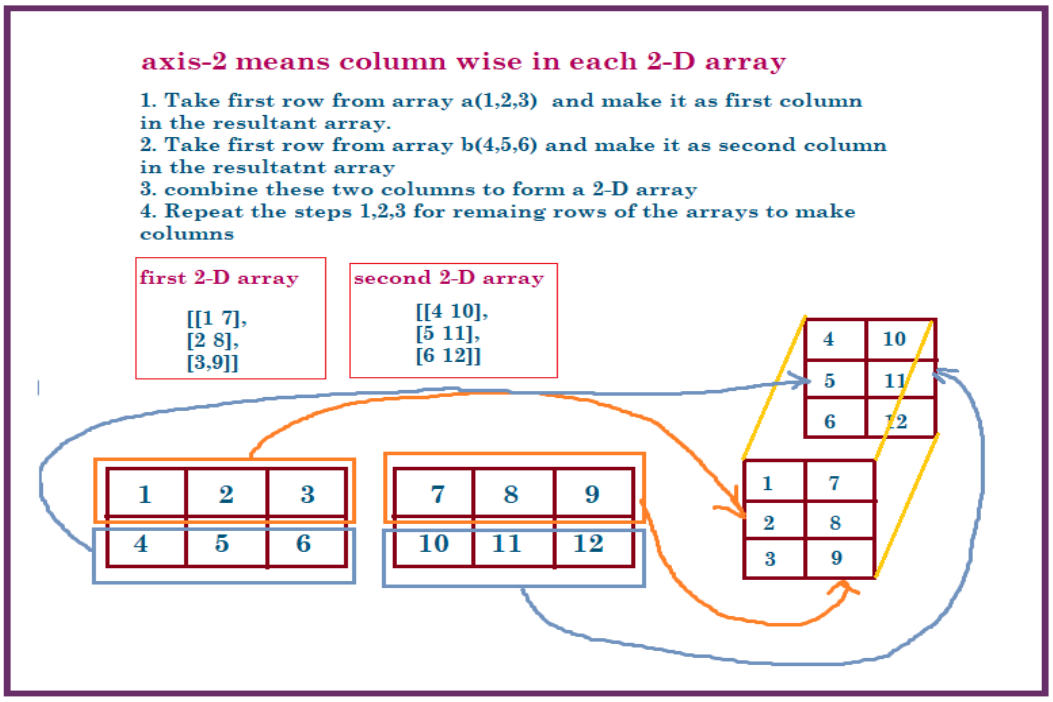
In [278]:

```
# stacking of 2-D arrays ==> axis=1
# axis-1 means row wise in each 2-D array
a = np.array([[1,2,3],[4,5,6]])
b = np.array([[7,8,9],[10,11,12]])
np.stack((a,b),axis=1)
```

Out[278]:

```
array([[[ 1,  2,  3],
        [ 7,  8,  9]],

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

3-D array stacking along axis-2



In [279]:

```
# stacking of 2-D arrays ==> axis=2
# axis-2 means column wise in each 2-D array
a = np.array([[1,2,3],[4,5,6]])
b = np.array([[7,8,9],[10,11,12]])
np.stack((a,b),axis=2)
```

Out[279]:

```
array([[[ 1,  7],
        [ 2,  8],
        [ 3,  9]],

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

In [280]:

```
# Demo of Stacking of Three 2-D arrays
a = np.arange(1,7).reshape(3,2)
b = np.arange(7,13).reshape(3,2)
c = np.arange(13,19).reshape(3,2)
print(f"array a :\n {a}")
print(f"array b :\n {b}")
print(f"array c :\n {c}")

# stacking along axis-0
# In 3-D array axis-0 means the number of 2-d arrays
axis0_stack = np.stack((a,b,c),axis=0)
print(f"Stacking three 2-D arrays along axis-0:\n {axis0_stack}")
```

```
array a :
 [[1 2]
 [3 4]
 [5 6]]
array b :
 [[ 7  8]
 [ 9 10]
 [11 12]]
array c :
 [[13 14]
 [15 16]
 [17 18]]
Stacking three 2-D arrays along axis-0:
 [[[ 1  2]
  [ 3  4]
  [ 5  6]]

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

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

In [281]:

```
# stacking along axis-1
# In 3-D array, axis-1 means the number of rows.
# Stacking row wise
axis1_stack = np.stack((a,b,c),axis=1)
print(f"Stacking three 2-D arrays along axis-1:\n {axis1_stack}")
```

```
Stacking three 2-D arrays along axis-1:
 [[[ 1  2]
  [ 7  8]
  [13 14]]

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

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

In [282]:

```
# stacking along axis-2
# in 3-D array axis-2 means the number of columns in every 2-D array.
# stacking column wise
axis2_stack = np.stack((a,b,c),axis=2)
print(f"Stacking three 2-D arrays along axis-2:\n {axis2_stack}")
```

```
Stacking three 2-D arrays along axis-2:
 [[[ 1  7 13]
  [ 2  8 14]]

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

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

Note:

- Reading of arrays row-wise
- arraning is based on the newly created array axis

Stacking Three 1-D array

```
In [283]:

a = np.arange(4)
b = np.arange(4,8)
c = np.arange(8,12)
print(f"array a :{a}")
print(f"array b :{b}")
print(f"array c :{c}")

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

# Based on axis-0:
# axis-0 in 2-D array means the number of rows
axis0_stack = np.stack((a,b,c),axis=0)
print(f"Stacking three 2-D arrays along axis-0:\n {axis0_stack}")

# Based on axis-1:
# axis-1 in 2-D array means the number of columns
axis1_stack = np.stack((a,b,c),axis=1)
print(f"Stacking three 2-D arrays along axis-1:\n {axis1_stack}")
```

```
array a :[0 1 2 3]
array b :[4 5 6 7]
array c :[ 8  9 10 11]
Stacking three 2-D arrays along axis-0:
 [[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
Stacking three 2-D arrays along axis-1:
 [[ 0  4  8]
 [ 1  5  9]
 [ 2  6 10]
 [ 3  7 11]]
```

cancatenate() Vs stack()

concateante()	stack()
Joining will be happened based on existing axis.	Joining will be happened based on new axis.
The dimension of newly created array is same as input array dimension.	The dimension of newly created array is one more than input array dimension.
To perform concatenation, all input arrays must have same dimension. The size of all dimensions except concatenation axis must be same.	To perform stack operation, compulsory all input arrays must have same shape.ie dimensions,sizes also needs to be same.

vstack()

- vstack--->vertical stack--->joining is always based on axis-0
- For 1-D arrays--->2-D array as output.
- For the remaining dimensions it acts as concatenate() along axis-0

Rules:

- The input arrays must have the same shape along all except first axis(axis-0)
- 1-D arrays must have the same size.
- The array formed by stacking the given arrays, will be at least 2-D.
- vstack() operation is equivalent to concatenation along the first axis after 1-D arrays of shape (N,) have been reshaped to (1,N).
- For 2-D or more dimension arrays, vstack() simply acts as concatenation wrt axis-0.

In [284]:

```
import numpy as np
help(np.vstack)
```

```
Help on function vstack in module numpy:

vstack(tup)
    Stack arrays in sequence vertically (row wise).

    This is equivalent to concatenation along the first axis after 1-D arrays
    of shape `(N,)` have been reshaped to `(1,N)`. Rebuilds arrays divided by
    `vsplit`.
```

For 1-D arrays

In [285]:

```
# vstack for 1-D arrays of same sizes
a = np.array([10,20,30,40])
b = np.array([50,60,70,80])
# a will be converted to shapes (1,4) and b will be converted to (1,4)
np.vstack((a,b))
```

Out[285]:

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

In [286]:

```
# vstack for 1-D arrays of different sizes
a = np.array([10,20,30,40])
b = np.array([50,60,70,80,90,100])
# a will be converted to shapes (1,4) and b will be converted to (1,6)
np.vstack((a,b))
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-286-7aa2c1e1cd41> in <module>
      3 b = np.array([50,60,70,80,90,100])
      4 # a will be converted to shapes (1,4) and b will be converted to (1,6
)
----> 5 np.vstack((a,b))

<__array_function__ internals> in vstack(*args, **kwargs)

F:\Users\Gopi\anaconda3\lib\site-packages\numpy\core\shape_base.py in vstack(
tup)
    281     if not isinstance(arrs, list):
    282         arrs = [arrs]
--> 283     return _nx.concatenate(arrs, 0)
    284
    285

<__array_function__ internals> in concatenate(*args, **kwargs)

ValueError: all the input array dimensions for the concatenation axis must ma
```

```
tch exactly, but along dimension 1, the array at index 0 has size 4 and the a
rray at index 1 has size 6
```

For 2-D arrays

In [287]:

```
# vstack of 2-D arrays. sizes of axis-1 should be same to perform vstack() ->
concatenation rule
# Here it is possible because sizes of axis-1 are same 3 and 3
# vstack() performed always along axis-0
a = np.arange(1,10).reshape(3,3)
b = np.arange(10,16).reshape(2,3)
np.vstack((a,b))
```

Out[287]:

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

In [288]:

```
# vstack of 2-D arrays. sizes of axis-1 should be same to perform vstack() ->
concatenation rule
# Here it is not possible because sizes of axis-1 are same 3 and 2
# vstack() performed always along axis-0
a = np.arange(1,10).reshape(3,3)
b = np.arange(10,16).reshape(3,2)
np.vstack((a,b))
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-288-4d14bef5f20b> in <module>
      4 a = np.arange(1,10).reshape(3,3)
      5 b = np.arange(10,16).reshape(3,2)
----> 6 np.vstack((a,b))

<__array_function__ internals> in vstack(*args, **kwargs)

F:\Users\Gopi\anaconda3\lib\site-packages\numpy\core\shape_base.py in vstack(
tup)
```

```
    281     if not isinstance(arrs, list):
    282         arrs = [arrs]
--> 283     return _nx.concatenate(arrs, 0)
    284
    285

<__array_function__ internals> in concatenate(*args, **kwargs)

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

For 3-D arrays

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

In [289]:

```
a = np.arange(1,25).reshape(2,3,4)
b = np.arange(25,49).reshape(2,3,4)
print(f"array a : \n {a}")
print(f"array b : \n {b}")
result = np.vstack((a,b))
print(f"Result of vstack  : \n {result}")
```

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

 [[13 14 15 16]
  [17 18 19 20]
  [21 22 23 24]]]
array b :
 [[[25 26 27 28]
  [29 30 31 32]
  [33 34 35 36]]

 [[37 38 39 40]
  [41 42 43 44]
  [45 46 47 48]]]
Result of vstack  :
 [[[ 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]]]
```

hstack()

- Exactly same as concatenate() but joining is always based on axis-1
- hstack--->horizontal stack--->column wise
- 1-D + 1-D --->1-D

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.

In [290]:

```
import numpy as np
help(np.hstack)
```

```
Help on function hstack in module numpy:

hstack(tup)
    Stack arrays in sequence horizontally (column wise).

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

For 1-D arrays

In [291]:

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

Out[291]:

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

For 2-D arrays

In [292]:

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

Out[292]:

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

In [293]:

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

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-293-0e470a6aee78> in <module>
      1 a = np.arange(1,7).reshape(2,3)
      2 b = np.arange(7,16).reshape(3,3)
----> 3 np.hstack((a,b))

<__array_function__ internals> in hstack(*args, **kwargs)

F:\Users\Gopi\anaconda3\lib\site-packages\numpy\core\shape_base.py in hstack(
tup)
    344         return _nx.concatenate(arrs, 0)
    345     else:
```

```
--> 346         return _nx.concatenate(arrs, 1)
    347
    348

<__array_function__ internals> in concatenate(*args, **kwargs)

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

dstack()

- dstack() --->depth/height stack --->concatenation based on axis-2
- 1-D and 2-D arrays will be converted to 3-D array
- The result is minimum 3-D array

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.

In [294]:

```
import numpy as np
help(np.dstack)
```

```
Help on function dstack in module numpy:

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

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

In [295]:

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

Out[295]:

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

In [296]:

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

Out[296]:

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

       [[2, 3]],

       [[3, 4]]])
```

Summary of joining of nd arrays:

- **concatenate()** ==> Join a sequence of arrays along an existing axis.
- **stack()** => Join a sequence of arrays along a new axis.
- **vstack()** => Stack arrays in sequence vertically according to first axis (axis-0).
- **hstack()** => Stack arrays in sequence horizontally according to second axis(axis-1).
- **dstack()** ==> Stack arrays in sequence depth wise according to third axis(axis-2).

Chapter-11
Splitting of arrays

Splitting of arrays

We can perform split operation on ndarrays using the following functions

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

We will get only **views**, but **not copies** because the data is not going to be changed

split()

In [297]:

```
import numpy as np
help(np.split)
```

```
Help on function split in module numpy:

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

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.

split() based on sections

- We can split arrays based on sections or indices.
- If we split based on sections, the sizes of sub-arrays should be equal.
- If we split based on indices, then the sizes of sub-arrays need not be the same

1-D arrays (axis=0)

In [298]:

```
a = np.arange(1,10)
sub_arrays = np.split(a,3)
print(f"array a : {a}")
print(f"Type of sub_arrays :{type(sub_arrays)}")
print(f"sub_arrays : {sub_arrays}")
```

```
array a : [1 2 3 4 5 6 7 8 9]
Type of sub_arrays :<class 'list'>
sub_arrays : [array([1, 2, 3]), array([4, 5, 6]), array([7, 8, 9])]
```

In [299]:

```
# If dividing array into equal number of specified sections is not possible, then we will get error.
np.split(a,4)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-299-821b8595e272> in <module>
      1 # If dividing array into equal number of specified sections is not po
ssible, then we will get error.
----> 2 np.split(a,4)

<__array_function__ internals> in split(*args, **kwargs)

F:\Users\Gopi\anaconda3\lib\site-packages\numpy\lib\shape_base.py in split(ar
y, indices_or_sections, axis)
    870         N = ary.shape[axis]
    871         if N % sections:
--> 872             raise ValueError(
    873                 'array split does not result in an equal division') f
rom None
    874     return array_split(ary, indices_or_sections, axis)

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

2-D arrays (axis=0 ==> Vertical split)

- splitting is based on axis-0 bydefault. ie row wise split(vertical split)
- We can also split based on axis-1. column wise split (horizontal split)

In [300]:

```
# split based on default axis i.e., axis-0 (Vertical Split)
a = np.arange(1,25).reshape(6,4)
result_3sections = np.split(a,3) # dividing 3 sections vertically
print(f"array a : \n {a}")
print(f"splitting the array into 3 sections along axis-0 : \n {result_3sections}")
# Note: Here we can use various possible sections: 2,3,6
```

```
array a :
 [[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]
 [13 14 15 16]
 [17 18 19 20]
 [21 22 23 24]]
splitting the array into 3 sections along 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]])]
```

In [301]:

```
# Note: Here we can use various possible sections: 2,3,6
result_2sections = np.split(a,2,axis=0)
result_6sections = np.split(a,6,axis=0)
print(f"splitting the array into 2 sections along axis-0 : \n {result_2sections}")
print(f"splitting the array into 6 sections along axis-0 : \n {result_6sections}")
```

```
splitting the array into 2 sections along axis-0 :
 [array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12]]), array([[13, 14, 15, 16],
       [17, 18, 19, 20],
       [21, 22, 23, 24]])]
splitting the array into 6 sections along axis-0 :
 [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]])]
```

2-D arrays (axis=1 ==> Horizontal split)

In [302]:

```
# split based on axis-1  (horizontal split)
# for the shape(6,4) we can perform 4 or 2 sections
a = np.arange(1,25).reshape(6,4)
result_2sections = np.split(a,2,axis=1) # dividing 2 sections horizontally
result_4sections = np.split(a,4,axis=1) # dividing 4 sections horizontally

print(f"splitting the array into 2 sections along axis-0 : \n {result_2sections}")
print(f"splitting the array into 4 sections along axis-0 : \n {result_4sections}")
```

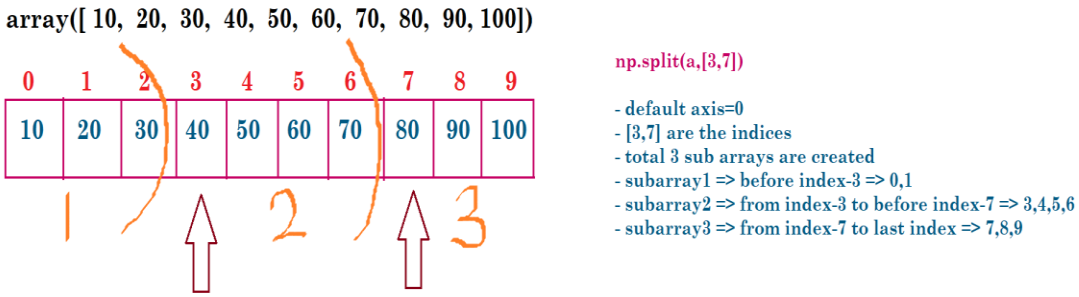
```
splitting the array into 2 sections along axis-0 :
 [array([[ 1,  2],
       [ 5,  6],
       [ 9, 10],
       [13, 14],
       [17, 18],
       [21, 22]]), array([[ 3,  4],
       [ 7,  8],
       [11, 12],
       [15, 16],
       [19, 20],
       [23, 24]])]
splitting the array into 4 sections along axis-0 :
 [array([[ 1],
       [ 5],
       [ 9],
       [13],
       [17],
       [21]]), array([[ 2],
       [ 6],
       [10],
       [14],
       [18],
       [22]]), array([[ 3],
       [ 7],
       [11],
       [15],
       [19],
       [23]]), array([[ 4],
       [ 8],
```

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

split() based on indices

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

1-D arrays(axis=0)



In [303]:

```
# splitting the 1-D array based on indices
a = np.arange(10,101,10)
result = np.split(a,[3,7])
print(f"array a : {a}")
print(f"splitting the 1-D array based on indices : \n {result}")
```

```
array a : [ 10  20  30  40  50  60  70  80  90 100]
splitting the 1-D array based on indices :
 [array([10, 20, 30]), array([40, 50, 60, 70]), array([ 80,  90, 100])]
```

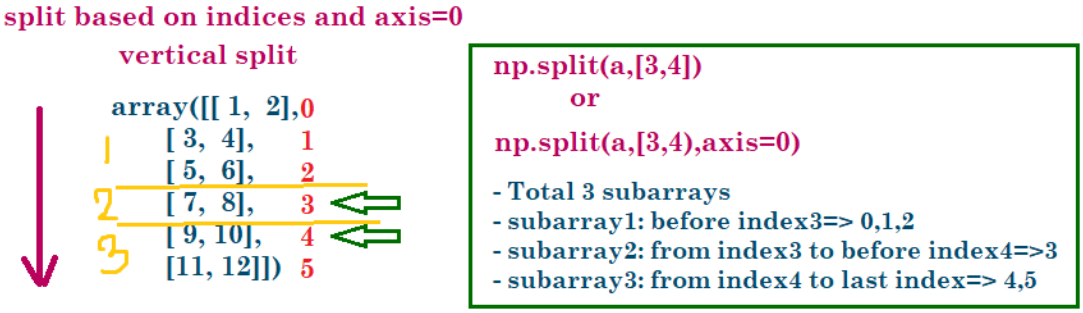
In [304]:

```
# splitting the 1-D array based on indices
a = np.arange(10,101,10)
result = np.split(a,[2,5,7])
# [2,5,7] ==> 4 subarrays
# subarray-1 : before index-2 ==> 0,1
# subarray-2 : from index-2 to before index-5 ==> 2,3,4
# subarray-3 : from index-5 to before index-7 ==> 5,6
# subarray-4 : from index-7 to last index ==> 7,8,9
```

```
print(f"array a : {a}")
print(f"splitting the 1-D array based on indices : \n {result}")
```

```
array a : [ 10  20  30  40  50  60  70  80  90 100]
splitting the 1-D array based on indices :
 [array([10, 20]), array([30, 40, 50]), array([60, 70]), array([ 80,  90, 100
])]
```

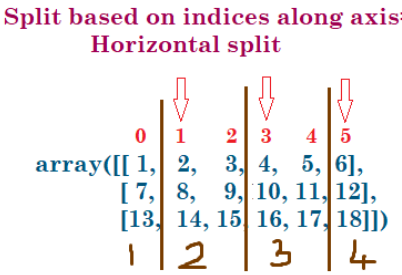
2-D arrays(axis=0)




```
In [305]:

# splitting 2-D arrays based on indices along axis=0
a = np.arange(1,13).reshape(6,2)
result = np.split(a,[3,4])
print(f"array a : \n {a}")
print(f"resultant array after vertical split  : \n {result}")
```

```
array a :
 [[ 1  2]
 [ 3  4]
 [ 5  6]
 [ 7  8]
 [ 9 10]
 [11 12]]
resultant array after vertical split  :
 [array([[1, 2],
       [3, 4],
       [5, 6]]), array([[7, 8]]), array([[ 9, 10],
       [11, 12]])]
```

2-D arrays(axis=1)




```
np.split(a,[1,3,5],axis=1)

- for [1,3,5] 4 subarrays are created
- subarray1: before index1=> 0
- subarray2: from index1 to before index3=>1,2
- subarray3: from index 3 to before index5=> 3,4
- subarray4: from index5 to last index => 5
```

In [306]:

```
a = np.arange(1,19).reshape(3,6)
result = np.split(a,[1,3,5],axis=1)
print(f"array a : \n {a}")
print(f"resultant array after horizontal split  : \n {result}")
```

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

In [307]:

```
a = np.arange(1,19).reshape(3,6)
result = np.split(a,[2,4,4],axis=1)
# [2,4,4] => 4 subarrays are created
# subarray1:  2 ==> before index-2 ==> 0,1
# subarray2:  4 ==> from index-2 to before index-4 ==> 2,3
# subarray3:  4 ==> from index-4 to before index-4 ==> empty array
# subarray4:    ==> from index-4 to last index ==> 4,5
```

```
print(f"array a : \n {a}")
print(f"resultant array after horizontal split  : \n {result}")
print(f"first subarray : \n {result[0]}")
print(f"second subarray : \n {result[1]}")
print(f"third subarray : \n {result[2]}")
print(f"fourth subarray : \n {result[3]}")
```

```
array a :
 [[ 1  2  3  4  5  6]
 [ 7  8  9 10 11 12]
 [13 14 15 16 17 18]]
resultant array after horizontal split  :
 [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]])]
first subarray :
 [[ 1  2]
 [ 7  8]
 [13 14]]
second subarray :
 [[ 3  4]
 [ 9 10]
 [15 16]]
third subarray :
 []
fourth subarray :
 [[ 5  6]
 [11 12]
 [17 18]]
```

In [308]:

```
a = np.arange(1,19).reshape(3,6)
result = np.split(a,[0,2,6],axis=1)
# [0,2,6] => 4 subarrays are created
# subarray1:  0 ==> before index-0 ==> empty
# subarray2:  2 ==> from index-0 to before index-2 ==> 0,1
# subarray3:  6 ==> from index-2 to before index-6 ==> 2,3,4,5
# subarray4:   ==> from index-6 to last index ==> empty
```

```
print(f"array a : \n {a}")
print(f"resultant array after horizontal split  : \n {result}")
print(f"first subarray : \n {result[0]}")
print(f"second subarray : \n {result[1]}")
print(f"third subarray : \n {result[2]}")
print(f"fourth subarray : \n {result[3]}")
```

```
array a :
 [[ 1  2  3  4  5  6]
 [ 7  8  9 10 11 12]
 [13 14 15 16 17 18]]
resultant array after horizontal split  :
 [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)]
first subarray :
 []
second subarray :
 [[ 1  2]
 [ 7  8]
 [13 14]]
third subarray :
 [[ 3  4  5  6]
 [ 9 10 11 12]
 [15 16 17 18]]
fourth subarray :
 []
```

In [309]:

```
a = np.arange(1,19).reshape(3,6)
result = np.split(a,[1,5,3],axis=1)
# [1,5,3] => 4 subarrays are created
# subarray1:  1 ==> before index-1 ==> 0
# subarray2:  5 ==> from index-1 to before index-5 ==> 1,2,3,4
# subarray3:  3 ==> from index-5 to before index-3 ==> empty
# subarray4:   ==> from index-3 to last index ==> 3,4,5
print(f"array a : \n {a}")
print(f"resultant array after horizontal split  : \n {result}")
print(f"first subarray : \n {result[0]}")
print(f"second subarray : \n {result[1]}")
```

```
print(f"third subarray : \n {result[2]}")
print(f"fourth subarray : \n {result[3]}")
```

```
array a :
 [[ 1  2  3  4  5  6]
 [ 7  8  9 10 11 12]
 [13 14 15 16 17 18]]
resultant array after horizontal split  :
 [array([[ 1],
       [ 7],
       [13]]), array([[ 2,  3,  4,  5],
       [ 8,  9, 10, 11],
       [14, 15, 16, 17]]), array([], shape=(3, 0), dtype=int32), array([[ 4,  5,  6],
       [10, 11, 12],
       [16, 17, 18]])]
first subarray :
 [[ 1]
 [ 7]
 [13]]
second subarray :
 [[ 2  3  4  5]
 [ 8  9 10 11]
 [14 15 16 17]]
third subarray :
 []
fourth subarray :
 [[ 4  5  6]
 [10 11 12]
 [16 17 18]]
```

vsplit()

- vsplit means vertical split means row wise split
- split is based on axis-0

In [310]:

```
import numpy as np
help(np.vsplit)
```

```
Help on function vsplit in module numpy:

vsplit(ary, indices_or_sections)
    Split an array into multiple sub-arrays vertically (row-wise).
```

1-D arrays

- To use vsplit, input array should be atleast 2-D array
- It is not possible to split 1-D array vertically.

In [311]:

```
a = np.arange(10)
np.vsplit(a,2)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-311-43d15b9a4cce> in <module>
      1 a = np.arange(10)
----> 2 np.vsplit(a,2)

<__array_function__ internals> in vsplit(*args, **kwargs)

F:\Users\Gopi\anaconda3\lib\site-packages\numpy\lib\shape_base.py in vsplit(a
ry, indices_or_sections)
    988     """
    989     if _nx.ndim(ary) < 2:
--> 990         raise ValueError('vsplit only works on arrays of 2 or more di
mensions')
    991     return split(ary, indices_or_sections, 0)
    992

ValueError: vsplit only works on arrays of 2 or more dimensions
```

2-D arrays

In [312]:

```
a = np.arange(1,13).reshape(6,2)
np.vsplit(a,2)
```

Out[312]:

```
[array([[1, 2],
        [3, 4],
        [5, 6]]),
 array([[ 7,  8],
        [ 9, 10],
        [11, 12]])]
```

In [313]:

```
np.vsplit(a,3)
```

Out[313]:

```
[array([[1, 2],
        [3, 4]]),
 array([[5, 6],
        [7, 8]]),
 array([[ 9, 10],
        [11, 12]])]
```

In [314]:

```
np.vsplit(a,6)
```

Out[314]:

```
[array([[1, 2]]),
 array([[3, 4]]),
 array([[5, 6]]),
 array([[7, 8]]),
 array([[ 9, 10]]),
 array([[11, 12]])]
```

In [315]:

```
# vsplit() based on indices:
a = np.arange(1,13).reshape(6,2)
np.vsplit(a,[3,4])
```

Out[315]:

```
[array([[1, 2],
        [3, 4],
        [5, 6]]),
 array([[7, 8]]),
 array([[ 9, 10],
        [11, 12]])]
```

hsplit()

- hsplit--->means horizontal split(column wise)
- split will be happend based on 2nd axis (axis-1)

In [316]:

```
import numpy as np
help(np.hsplit)
```

```
Help on function hsplit in module numpy:

hsplit(ary, indices_or_sections)
    Split an array into multiple sub-arrays horizontally (column-wise).
```

1-D arrays

In [317]:

```
a = np.arange(10)
np.hsplit(a,2)
```

Out[317]:

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

2-D arrays

- Based on axis-1 only

In [318]:

```
a = np.arange(1,13).reshape(3,4)
np.hsplit(a,2)
```

Out[318]:

```
[array([[ 1,  2],
        [ 5,  6],
        [ 9, 10]]),
 array([[ 3,  4],
        [ 7,  8],
        [11, 12]])]
```

In [319]:

```
# hsplit() based on indices:
a = np.arange(10,101,10)
np.hsplit(a,[2,4])
```

Out[319]:

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

In [320]:

```
a = np.arange(24).reshape(4,6)
np.hsplit(a,[2,4])
```

Out[320]:

```
[array([[ 0,  1],
        [ 6,  7],
        [12, 13],
        [18, 19]]),
 array([[ 2,  3],
        [ 8,  9],
        [14, 15],
        [20, 21]]),
 array([[ 4,  5],
        [10, 11],
        [16, 17],
        [22, 23]])]
```

dsplit()

- dsplit --->means depth split
- splitting based on 3rd axis(axis-2).

In [321]:

```
import numpy as np
help(np.dsplit)
```

```
Help on function dsplit in module numpy:

dsplit(ary, indices_or_sections)
    Split array into multiple sub-arrays along the 3rd axis (depth).
```

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 eq
ual to 3

In [322]:

```
a = np.arange(24).reshape(2,3,4)
a
```

Out[322]:

```
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]]])
```

In [323]:

```
# dsplit based on sections
np.dsplit(a,2)
```

Out[323]:

```
[array([[[ 0,  1],
         [ 4,  5],
         [ 8,  9]],

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

        [[14, 15],
         [18, 19],
         [22, 23]]])]
```

In [324]:

```
# dsplit based on indices
np.dsplit(a,[1,3])
```

Out[324]:

```
[array([[[ 0],
         [ 4],
         [ 8]],

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

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

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

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.

In [325]:

```
import numpy as np
help(np.array_split)
```

```
Help on function array_split in module numpy:

array_split(ary, 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

In [326]:

```
#  x % n sub-arrays of size x//n + 1
#  and the rest of size x//n
# eg-1:
# 10 elements --->3 sections
# 10%3(1) sub-arrays of size 10//3+1(4)
# and the rest(2) of size 10//3 (3)
# 1 sub-array of size 4 and the rest of size 3
#(4,3,3)
a = np.arange(10,101,10)
np.array_split(a,3)
```

Out[326]:

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

In [327]:

```
# Eg: 2
# 11 elements   3 sections

# it returns x % n (11%3=2)sub-arrays of size x//n + 1(11//3+1=4)
# and the rest(1) of size x//n.(11//3=3)
# (4,4,3)
a = np.arange(11)
np.array_split(a,3)
```

Out[327]:

```
[array([0, 1, 2, 3]), array([4, 5, 6, 7]), array([ 8,  9, 10])]
```

In [328]:

```
# 2-D array
# x=6 n=4
# x % n sub-arrays of size x//n + 1-->2 sub-arrays of size:2
# rest of size x//n.--->2 sub-arrays of size:1
# 2,2,1,1,
```

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

Out[328]:

```
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]])
```

In [329]:

np.array_split(a,4)

Out[329]:

```
[array([[0, 1, 2, 3],
        [4, 5, 6, 7]]),
 array([[ 8,  9, 10, 11],
        [12, 13, 14, 15]]),
 array([[16, 17, 18, 19]]),
 array([[20, 21, 22, 23]])]
```

Summary of split methods:

- **split()** ==>Split an array into multiple sub-arrays of equal size. Raise error if an equal division is cannot be made
- **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.

Chapter-12
Sorting elements of nd arrays

Sorting elements of nd arrays

- We can sort elements of nd array.
- numpy module contains sort() function.
- The default sorting algorithm is `quicksort` and it is `Ascending order`
- We can also specify mergesort, heapsort etc
- For numbers-->Ascending order
- For Strings-->alphabetical order

In [330]:

```
import numpy as np
help(np.sort)
```

```
Help on function sort in module numpy:

sort(a, axis=-1, kind=None, order=None)
    Return a sorted copy of an array.
```

1-D arrays

In [331]:

```
# 1-D arrays
a = np.array([70,20,60,10,50,40,30])
sorted_array = np.sort(a)
print(f"Original array a : {a}")
print(f"Sorted array(Ascending by default) : {sorted_array}")
```

```
Original array a : [70 20 60 10 50 40 30]
Sorted array(Ascending by default) : [10 20 30 40 50 60 70]
```

In [332]:

```
# Descending the 1-D arrays
# 1st way :: np.sort(a)[::-1]
a = np.array([70,20,60,10,50,40,30])
sorted_array = np.sort(a)[::-1]
```

```
print(f"Original array a : {a}")
print(f"Sorted array(Descending) : {sorted_array}")
```

```
Original array a : [70 20 60 10 50 40 30]
Sorted array(Descending) : [70 60 50 40 30 20 10]
```

In [333]:

```
# Descending the 1-D arrays
# 2nd way :: -np.sort(-a)
a = np.array([70,20,60,10,50,40,30])
-a
```

Out[333]:

```
array([-70, -20, -60, -10, -50, -40, -30])
```

In [334]:

```
np.sort(-a) # Ascending order
```

Out[334]:

```
array([-70, -60, -50, -40, -30, -20, -10])
```

In [335]:

```
-np.sort(-a) # Descending Order
```

Out[335]:

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

In [336]:

```
# To sort string elements in alphabetical order
a = np.array(['cat','rat','bat','vat','dog'])
ascending = np.sort(a)
descending = np.sort(a)[::-1]
# -np.sort(-a) ==> 2nd way is not possible
print(f"Original array a : {a}")
print(f"Sorted array(Ascending) : {ascending}")
print(f"Sorted array(Descending) : {descending}")
```

```
Original array a : ['cat' 'rat' 'bat' 'vat' 'dog']
Sorted array(Ascending) : ['bat' 'cat' 'dog' 'rat' 'vat']
Sorted array(Descending) : ['vat' 'rat' 'dog' 'cat' 'bat']
```

2-D arrays

- axis-0 --->the number of rows (axis = -2)
- axis-1--->the number of columns (axis= -1) ==> defautl value
- sorting is based on columns in 2-D arrays by default. Every 1-D array will be sorted.

In [337]:

```
a= np.array([[40,20,70],[30,20,60],[70,90,80]])
ascending = np.sort(a)
print(f"Original array a :\n {a}")
print(f"Sorted array(Ascending) : \n {ascending}")
```

```
Original array a :
 [[40 20 70]
 [30 20 60]
 [70 90 80]]
Sorted array(Ascending) :
 [[20 40 70]
 [20 30 60]
 [70 80 90]]
```

order parameter

- Use the `order` keyword to specify a field to use when sorting a structured array:

In [338]:

```
# creating the structured array
dtype = [('name', 'S10'), ('height', float), ('age', int)]
values = [('Gopie', 1.7, 45), ('Vikranth', 1.5, 38),('Sathwik', 1.8, 28)]
a = np.array(values, dtype=dtype)
sort_height =  np.sort(a, order='height')
sort_age = np.sort(a,order='age')
print(f"Original Array :\n {a}")
print(f"Sorting based on height :\n {sort_height}")
print(f"Sorting based on age :\n {sort_age}")
```

```
Original Array :
 [(b'Gopie', 1.7, 45) (b'Vikranth', 1.5, 38) (b'Sathwik', 1.8, 28)]
Sorting based on height :
 [(b'Vikranth', 1.5, 38) (b'Gopie', 1.7, 45) (b'Sathwik', 1.8, 28)]
Sorting based on age :
 [(b'Sathwik', 1.8, 28) (b'Vikranth', 1.5, 38) (b'Gopie', 1.7, 45)]
```

In [339]:

```
# Sort by age, then height if ages are equal
dtype = [('name', 'S10'), ('height', float), ('age', int)]
values = [('Gopie', 1.7, 45), ('Vikranth', 1.5, 38),('Sathwik', 1.8, 28),('Rudra', 1.5, 28)]
a = np.array(values, dtype=dtype)
sort_age_height = np.sort(a, order=['age', 'height'])
print(f"Original Array :\n {a}")
print(f"Sorting based on height :\n {sort_age_height}")
```

```
Original Array :
 [(b'Gopie', 1.7, 45) (b'Vikranth', 1.5, 38) (b'Sathwik', 1.8, 28)
 (b'Rudra', 1.5, 28)]
Sorting based on height :
 [(b'Rudra', 1.5, 28) (b'Sathwik', 1.8, 28) (b'Vikranth', 1.5, 38)
 (b'Gopie', 1.7, 45)]
```

Chapter-13
Searching elements of ndarray

Searching elements of ndarray

- We can search elements of ndarray by using where() function.

where(condition, [x, y])

- If we specify only the condition, then it will return the indices of the elements which satisfies the condition.
- If we provide condition,x,y then The elements which satisfies the condition will be replaced with x and remaining elements will be replaced with y
- where() function does not return the elements. It returns only the indices
- it will act as replacement operator also.
- The functionality is just similar to ternary operator when it acts as replacement operator

In [340]:

```
import numpy as np
help(np.where)
```

```
Help on function where in module numpy:

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

    Return elements chosen from `x` or `y` depending on `condition`.
```

where() function

In [341]:

```
# 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 # element 7 is available at 2 and 4 indices
```

Out[341]:

```
(array([2, 4], dtype=int64),)
```

In [342]:

```
# Find indices 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
```

Out[342]:

```
(array([0, 1, 2, 4, 5, 9], dtype=int64),)
```

Finding the elements directly

We can get the elements directly in 2 ways

1. using where() function
2. using condition based selection

where() function

In [343]:

```
# to get the odd numbers
a = np.array([3,5,7,6,7,9,4,6,10,15])
indices = np.where(a%2!=0)
a[indices]
```

Out[343]:

```
array([ 3,  5,  7,  7,  9, 15])
```

conditional based selection

In [344]:

```
# to get the odd numbers
a = np.array([3,5,7,6,7,9,4,6,10,15])
a[a%2!=0]
```

Out[344]:

```
array([ 3,  5,  7,  7,  9, 15])
```

In [345]:

```
# 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.
# 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
```

Out[345]:

```
array([7777, 7777, 7777, 8888, 7777, 7777, 8888, 8888, 8888, 7777])
```

In [346]:

```
# 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
```

Out[346]:

```
array([9999, 9999, 9999,    6, 9999, 9999,    4,    6,   10, 9999])
```

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

2-D arrays

- It will return the 2 arrays.
- First array is row indices
- Second array is column indices

In [347]:

```
# to find the indices of the elements where elements are divisible by 5
a = np.arange(12).reshape(4,3)
np.where(a%5==0)
```

Out[347]:

```
(array([0, 1, 3], dtype=int64), array([0, 2, 1], dtype=int64))
```

- The first array array([0, 1, 3] represents the row indices

- The second array array([0, 2, 1] represents the column indices
- The required elements present at (0,0),(1,2) and (3,1) index places.

In [348]:

```
# we can perform replacement on 2-D arrays
a = np.arange(12).reshape(4,3)
np.where(a%5==0,9999,a)
```

Out[348]:

```
array([[9999,    1,    2],
       [   3,    4, 9999],
       [   6,    7,    8],
       [   9, 9999,   11]])
```

searchsorted() function

- Internally this function will use Binary Search algorithm. Hence we can call this function only for sorted arrays.
- If the array is not sorted then we will get abnormal results.
- Complexicity of the Binary search algorithm is O(log n)
- It will return insertion point(i.e., index) of the given element

In [349]:

```
import numpy as np
help(np.searchsorted)
```

```
Help on function searchsorted in module numpy:

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

In [350]:

```
# to find the insertion point of 6 from left
a = np.arange(0,31,5)
np.searchsorted(a,6)
```

Out[350]:

```
2
```

Note

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

In [351]:

```
# to find the insertion point of 6 from right
a = np.arange(0,31,5)
print(f"Array a :\n {a}")
np.searchsorted(a,6,side='right')
```

```
Array a :
 [ 0  5 10 15 20 25 30]
```

Out[351]:

```
2
```

In [352]:

```
# to find the insetion point from left and right
a = np.array([3,5,7,6,7,9,4,10,15,6])
# first sort the elements
a = np.sort(a)

# insertion point from left(default)
left = np.searchsorted(a,6)
# insertion point from right
right = np.searchsorted(a,6,side='right')

print(f"The original array : {a}")
print(f"Insertion point for 6 from left : {left}")
print(f"Insertion point for 6 from right : {right}")
```

```
The original array : [ 3  4  5  6  6  7  7  9 10 15]
Insertion point for 6 from left : 3
Insertion point for 6 from right : 5
```

Summary:

- **sort()** ==> To sort given array
- **where()** ==> To perform search and replace operation
- **searchsorted()** ==> To identify insertion point in the given sorted array

Chapter-14
How to insert elements into ndarray

How to insert elements into ndarray

- **insert()** ==> inserting the element at the required position
- **append()** ==> inserting the element at the end

insert()

In [353]:

```
import numpy as np
help(np.insert)
```

```
Help on function insert in module numpy:

insert(arr, obj, values, axis=None)
    Insert values along the given axis before the given indices.
```

In [354]:

```
# 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' are inserted.
#    values--->Values to insert into array.
#    axis ---->Axis along which to insert 'values' .
```

1-D arrays

In [355]:

```
#  To insert 7777 before index 2
a = np.arange(10)
b = np.insert(a,2,7777)
print(f"array a : {a}")
print(f"array b : {b}")
```

```
array a : [0 1 2 3 4 5 6 7 8 9]
array b : [   0    1 7777    2    3    4    5    6    7    8    9]
```

In [356]:

```
# To insert 7777 before indexes 2 and 5.
a = np.arange(10)
b = np.insert(a,[2,5],7777)
print(f"array a : {a}")
print(f"array b : {b}")
```

```
array a : [0 1 2 3 4 5 6 7 8 9]
array b : [   0    1 7777    2    3    4 7777    5    6    7    8    9]
```

In [357]:

```
# To insert 7777 before index 2 and 8888 before index 5?
a = np.arange(10)
b = np.insert(a,[2,5],[7777,8888])
print(f"array a : {a}")
print(f"array b : {b}")
```

```
array a : [0 1 2 3 4 5 6 7 8 9]
array b : [   0    1 7777    2    3    4 8888    5    6    7    8    9]
```

In [358]:

```
# shape mismatch
a = np.arange(10)
b = np.insert(a,[2,5],[7777,8888,9999])
print(f"array a : {a}")
print(f"array b : {b}")
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-358-735011b8a30e> in <module>
      1 # shape mismatch
      2 a = np.arange(10)
----> 3 b = np.insert(a,[2,5],[7777,8888,9999])
      4 print(f"array a : {a}")
      5 print(f"array b : {b}")

<__array_function__ internals> in insert(*args, **kwargs)

F:\Users\Gopi\anaconda3\lib\site-packages\numpy\lib\function_base.py in inser
t(arr, obj, values, axis)
   4676     slobj[axis] = indices
```

```
   4677     slobj2[axis] = old_mask
-> 4678     new[tuple(slobj)] = values
   4679     new[tuple(slobj2)] = arr
   4680

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

In [359]:

```
# shape mismatch
a = np.arange(10)
b = np.insert(a,[2,5,7],[7777,8888])
print(f"array a : {a}")
print(f"array b : {b}")
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-359-d8cdd8c18d4d> in <module>
      1 # shape mismatch
      2 a = np.arange(10)
----> 3 b = np.insert(a,[2,5,7],[7777,8888])
      4 print(f"array a : {a}")
      5 print(f"array b : {b}")

<__array_function__ internals> in insert(*args, **kwargs)

F:\Users\Gopi\anaconda3\lib\site-packages\numpy\lib\function_base.py in inser
t(arr, obj, values, axis)
   4676     slobj[axis] = indices
   4677     slobj2[axis] = old_mask
-> 4678     new[tuple(slobj)] = values
   4679     new[tuple(slobj2)] = arr
   4680

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

In [360]:

```
a = np.arange(10)
b = np.insert(a,[2,5,5],[777,888,999])
```

```
print(f"array a : {a}")
print(f"array b : {b}")
```

```
array a : [0 1 2 3 4 5 6 7 8 9]
array b : [  0   1 777   2   3   4 888 999   5   6   7   8   9]
```

```
In [361]:

# IndexError
a = np.arange(10)
b = np.insert(a,25,7777)
print(f"array a : {a}")
print(f"array b : {b}")
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-361-423ff78357e9> in <module>
      1 # IndexError
      2 a = np.arange(10)
----> 3 b = np.insert(a,25,7777)
      4 print(f"array a : {a}")
      5 print(f"array b : {b}")

<__array_function__ internals> in insert(*args, **kwargs)

F:\Users\Gopi\anaconda3\lib\site-packages\numpy\lib\function_base.py in inser
t(arr, obj, values, axis)
   4630         index = indices.item()
   4631         if index < -N or index > N:
-> 4632             raise IndexError(
   4633                 "index %i is out of bounds for axis %i with "
   4634                 "size %i" % (obj, axis, N))

IndexError: index 25 is out of bounds for axis 0 with size 10
```

Note

- All the insertion points(indices) are identified at the beginning of the insert operation
- Array should contain 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.

In [362]:

```
# the original array contains int values. If inserted value is float then the float value
# is  converted to the array type i.e., int. There may be data loss in this scenario
a = np.arange(10)
b = np.insert(a,2,123.456)
print(f"array a : {a}")
print(f"array b : {b}")
```

```
array a : [0 1 2 3 4 5 6 7 8 9]
array b : [  0   1 123   2   3   4   5   6   7   8   9]
```

In [363]:

```
a = np.arange(10)
b = np.insert(a,2,True)
print(f"array a : {a}")
print(f"array b : {b}")
```

```
array a : [0 1 2 3 4 5 6 7 8 9]
array b : [0 1 1 2 3 4 5 6 7 8 9]
```

In [364]:

```
a = np.arange(10)
b = np.insert(a,2,'GK')
print(f"array a : {a}")
print(f"array b : {b}")
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-364-57774f62be41> in <module>
      1 a = np.arange(10)
----> 2 b = np.insert(a,2,'GK')
      3 print(f"array a : {a}")
      4 print(f"array b : {b}")

<__array_function__ internals> in insert(*args, **kwargs)

F:\Users\Gopi\anaconda3\lib\site-packages\numpy\lib\function_base.py in inser
t(arr, obj, values, axis)
   4638         # There are some object array corner cases here, but we canno
t avoid
   4639         # that:
-> 4640         values = array(values, copy=False, ndmin=arr.ndim, dtype=arr.
```

```
dtype)
   4641         if indices.ndim == 0:
   4642             # broadcasting is very different here, since a[:,0,:] = .
.. behaves

ValueError: invalid literal for int() with base 10: 'GK'
```

In [365]:

```
a = np.arange(10)
b = np.insert(a,2,10+20j)
print(f"array a : {a}")
print(f"array b : {b}")
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-365-d9aabac22b2c> in <module>
      1 a = np.arange(10)
----> 2 b = np.insert(a,2,10+20j)
      3 print(f"array a : {a}")
      4 print(f"array b : {b}")

<__array_function__ internals> in insert(*args, **kwargs)

F:\Users\Gopi\anaconda3\lib\site-packages\numpy\lib\function_base.py in inser
t(arr, obj, values, axis)
   4638         # There are some object array corner cases here, but we canno
t avoid
   4639         # that:
-> 4640         values = array(values, copy=False, ndmin=arr.ndim, dtype=arr.
dtype)
   4641         if indices.ndim == 0:
   4642             # broadcasting is very different here, since a[:,0,:] = .
.. behaves

TypeError: can't convert complex to int
```

Summary for 1-D arrays while insertion

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

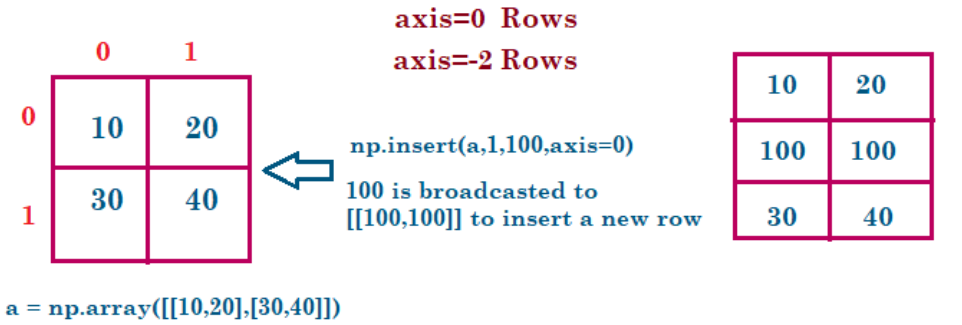
2-D arrays

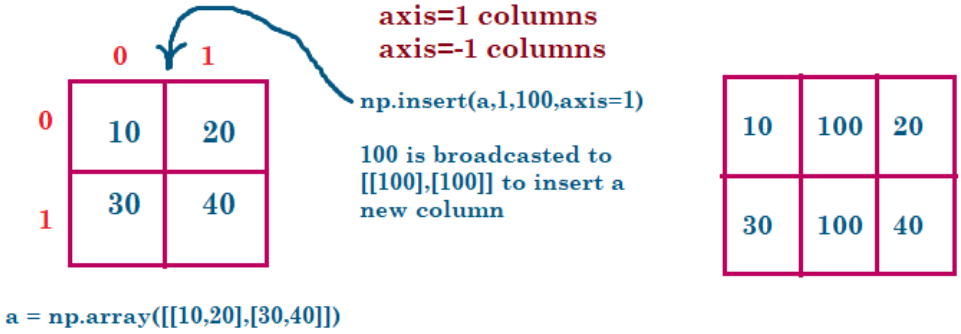
- If we are trying to insert elements into multi dimensional arrays, compulsory we have to provide axis
- If we are not providing axis value, then default value None will be considered
- In this case, array will be flatten to 1-D array and then insertion will be happend.
- axis=0 means rows (axis=-2)
- axis=1 means columns (axis=-1)

In [366]:

```
# if the axis is not defined for 2-D arrays, None is selected by default
# here the 2-D array flatten to 1-D array and insertion will be happened
a = np.array([[10,20],[30,40]])
b = np.insert(a,1,100)
print(f"array a :\n {a}")
print(f"array b : {b}")
```

```
array a :
 [[10 20]
 [30 40]]
array b : [ 10 100  20  30  40]
```





In [367]:

```
# insert the elements along the axis=0 or axis=-2 ==> rows are inserted
a = np.array([[10,20],[30,40]])
b = np.insert(a,1,100,axis=0)
c = np.insert(a,1,100,axis=-2)
print(f"array a :\n {a}")
print(f"array b :\n {b}")
print(f"array c :\n {c}")
```

```
array a :
 [[10 20]
 [30 40]]
array b :
 [[ 10  20]
 [100 100]
 [ 30  40]]
array c :
 [[ 10  20]
```

```
 [100 100]
 [ 30  40]]
```

In [368]:

```
# insert the elements along the axis=1 or axis=-1 ==> rows are inserted
a = np.array([[10,20],[30,40]])
b = np.insert(a,1,100,axis=1)
c = np.insert(a,1,100,axis=-1)
print(f"array a :\n {a}")
print(f"array b :\n {b}")
print(f"array c :\n {c}")
```

```
array a :
 [[10 20]
 [30 40]]
array b :
 [[ 10 100  20]
 [ 30 100  40]]
array c :
 [[ 10 100  20]
 [ 30 100  40]]
```

In [369]:

```
# to insert multiple rows
a = np.array([[10,20],[30,40]])
b = np.insert(a,1,[[100,200],[300,400]],axis=0)

print(f"array a :\n {a}")
print(f"array b :\n {b}")
```

```
array a :
 [[10 20]
 [30 40]]
array b :
 [[ 10  20]
 [100 200]
 [300 400]
 [ 30  40]]
```

In [370]:

```
# to insert multiple columns
a = np.array([[10,20],[30,40]])
b = np.insert(a,1,[[100,200],[300,400]],axis=1)

print(f"array a :\n {a}")
print(f"array b :\n {b}")
```

```
array a :
 [[10 20]
 [30 40]]
array b :
 [[ 10 100 300  20]
 [ 30 200 400  40]]
```

In [371]:

```
# ValueError
a = np.array([[10,20],[30,40]])
b =  np.insert(a,1,[100,200,300],axis=0)

print(f"array a :\n {a}")
print(f"array b :\n {b}")
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-371-926c0e00dba4> in <module>
      1 # ValueError
      2 a = np.array([[10,20],[30,40]])
----> 3 b =  np.insert(a,1,[100,200,300],axis=0)
      4
      5 print(f"array a :\n {a}")

<__array_function__ internals> in insert(*args, **kwargs)

F:\Users\Gopi\anaconda3\lib\site-packages\numpy\lib\function_base.py in inser
t(arr, obj, values, axis)
   4650         new[tuple(slobj)] = arr[tuple(slobj)]
   4651         slobj[axis] = slice(index, index+numnew)
-> 4652         new[tuple(slobj)] = values
   4653         slobj[axis] = slice(index+numnew, None)
   4654         slobj2 = [slice(None)] * ndim
```

```
ValueError: could not broadcast input array from shape (1,3) into shape (1,2)
```

append()

- By using insert() function, we can insert elements at our required index position.
- If we want to add elements always at end of the ndarray, then we have to go for append() function.

In [372]:

```
import numpy as np
help(np.append)
```

```
Help on function append in module numpy:

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

1-D arrays

- If appended element type is not same type of array, then elements conversion will be happend such that all elements of same common type.

In [373]:

```
# append 100 element to the existing array of int
a = np.arange(10)
b = np.append(a,100)
print(f"array a : {a}")
print(f"array b : {b}")
```

```
array a : [0 1 2 3 4 5 6 7 8 9]
array b : [  0   1   2   3   4   5   6   7   8   9 100]
```

In [374]:

```
# if the inserted element is not the type of the array then common type conversion
happened
a = np.arange(10)
b = np.append(a,20.5)
print(f"array a : {a}")
print(f"array b : {b}")
```

```
array a : [0 1 2 3 4 5 6 7 8 9]
array b : [ 0.   1.   2.   3.   4.   5.   6.   7.   8.   9.  20.5]
```

In [375]:

```
a = np.arange(10)
b = np.append(a,"GK")
print(f"array a : {a}")
print(f"array b : {b}")
```

```
array a : [0 1 2 3 4 5 6 7 8 9]
array b : ['0' '1' '2' '3' '4' '5' '6' '7' '8' '9' 'GK']
```

In [376]:

```
a = np.arange(10)
b = np.append(a,True)
print(f"array a : {a}")
print(f"array b : {b}")
```

```
array a : [0 1 2 3 4 5 6 7 8 9]
array b : [0 1 2 3 4 5 6 7 8 9 1]
```

In [377]:

```
a = np.arange(10)
b = np.append(a,20+15j)
print(f"array a : {a}")
print(f"array b : {b}")
```

```
array a : [0 1 2 3 4 5 6 7 8 9]
array b : [ 0. +0.j  1. +0.j  2. +0.j  3. +0.j  4. +0.j  5. +0.j  6. +0.j  7.
+0.j  8. +0.j  9. +0.j 20.+15.j]
```

2-D arrays

- If we are not specifying axis, then input array will be flatten to 1-D array and then append will be performed.
- If we are providing axis, then all the input arrays must have same number of dimensions, and same shape of provided axis.
- if axis-0 is taken then the obj should match with the number of columns of the input array
- if axis-1 is taken then the obj should match with the number of rows of the input array

In [378]:

```
# here axis is not specified so the default "None" will be taken and flatten to 1-D array & insertion
a = np.array([[10,20],[30,40]])
b = np.append(a,70)
print(f"array a :\n {a}")
print(f"array b : {b}")
```

```
array a :
 [[10 20]
 [30 40]]
array b : [10 20 30 40 70]
```

In [379]:

```
# axis=0 ==> along rows
# Value Error : i/p array is 2-D and appended array is 0-D
a = np.array([[10,20],[30,40]])
b = np.append(a,70,axis=0)  # appended is 0-D => 70
print(f"array a :\n {a}")
print(f"array b : {b}")
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-379-c2af6bdb12eb> in <module>
      2 # Value Error : i/p array is 2-D and appended array is 0-D
      3 a = np.array([[10,20],[30,40]])
----> 4 b = np.append(a,70,axis=0)  # appended is 0-D => 70
      5 print(f"array a :\n {a}")
      6 print(f"array b : {b}")

<__array_function__ internals> in append(*args, **kwargs)

F:\Users\Gopi\anaconda3\lib\site-packages\numpy\lib\function_base.py in append(arr, values, axis)
   4743         values = ravel(values)
   4744         axis = arr.ndim-1
-> 4745     return concatenate((arr, values), axis=axis)
   4746
   4747

<__array_function__ internals> in concatenate(*args, **kwargs)
```

```
ValueError: all the input arrays must have same number of dimensions, but the
array at index 0 has 2 dimension(s) and the array at index 1 has 0 dimension(
s)
```

In [380]:

```
# Value Error : i/p array is 2-D and appended array is 1-D
a = np.array([[10,20],[30,40]])
b = np.append(a,[70,80],axis=0)  # appended is 1-D => [70,80]
print(f"array a :\n {a}")
print(f"array b : {b}")
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-380-d1f0cddf5145> in <module>
      1 # Value Error : i/p array is 2-D and appended array is 1-D
      2 a = np.array([[10,20],[30,40]])
----> 3 b = np.append(a,[70,80],axis=0)  # appended is 1-D => [70,80]
      4 print(f"array a :\n {a}")
      5 print(f"array b : {b}")

<__array_function__ internals> in append(*args, **kwargs)

F:\Users\Gopi\anaconda3\lib\site-packages\numpy\lib\function_base.py in appen
d(arr, values, axis)
   4743         values = ravel(values)
   4744         axis = arr.ndim-1
-> 4745     return concatenate((arr, values), axis=axis)
   4746
   4747

<__array_function__ internals> in concatenate(*args, **kwargs)

ValueError: all the input arrays must have same number of dimensions, but the
array at index 0 has 2 dimension(s) and the array at index 1 has 1 dimension(
s)
```

In [381]:

```
# i/p array is 2-D and appended array is 2-D
a = np.array([[10,20],[30,40]])
b = np.append(a,[[70,80]],axis=0)  # appended is 2-D => [[70,80]]
```

```
print(f"array a :\n {a}")
print(f"array b :\n {b}")
```

```
array a :
 [[10 20]
 [30 40]]
array b :
 [[10 20]
 [30 40]
 [70 80]]
```

```
In [382]:

# axis=1 along colums
# Value Error : i/p array is Size 2 and appended element is size 1
a = np.array([[10,20],[30,40]])
b = np.append(a,[[70,80]],axis=1)  # appended is size 1 => [[70,80]]
print(f"array a :\n {a}")
print(f"array b : {b}")
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-382-469d4a0f90ba> in <module>
      2 # Value Error : i/p array is Size 2 and appended element is size 1
      3 a = np.array([[10,20],[30,40]])
----> 4 b = np.append(a,[[70,80]],axis=1)  # appended is size 1 => [[70,80]]
      5 print(f"array a :\n {a}")
      6 print(f"array b : {b}")

<__array_function__ internals> in append(*args, **kwargs)

F:\Users\Gopi\anaconda3\lib\site-packages\numpy\lib\function_base.py in appen
d(arr, values, axis)
   4743         values = ravel(values)
   4744         axis = arr.ndim-1
-> 4745     return concatenate((arr, values), axis=axis)
   4746
   4747

<__array_function__ internals> in concatenate(*args, **kwargs)

ValueError: all the input array dimensions for the concatenation axis must ma
```

```
tch exactly, but along dimension 0, the array at index 0 has size 2 and the a
rray at index 1 has size 1
```

In [383]:

```
a = np.array([[10,20],[30,40]])
b = np.append(a,[[70],[80]],axis=1)
print(f"array a :\n {a}")
print(f"array b :\n {b}")
```

```
array a :
 [[10 20]
 [30 40]]
array b :
 [[10 20 70]
 [30 40 80]]
```

In [384]:

```
# multiple columns
a = np.array([[10,20],[30,40]])
b =  np.append(a,[[70,80],[90,100]],axis=1)
print(f"array a :\n {a}")
print(f"array b :\n {b}")
```

```
array a :
 [[10 20]
 [30 40]]
array b :
 [[ 10  20  70  80]
 [ 30  40  90 100]]
```

In [385]:

```
# Consider the array?
a = np.arange(12).reshape(4,3)

# Which of the following operations will be performed successfully?
# A. np.append(a,[[10,20,30]],axis=0) #valid
# B. np.append(a,[[10,20,30]],axis=1) #invalid
# C. np.append(a,[[10],[20],[30]],axis=0) #invalid
# D. np.append(a,[[10],[20],[30],[40]],axis=1) #valid
# E. np.append(a,[[10,20,30],[40,50,60]],axis=0) #valid
# F. np.append(a,[[10,20],[30,40],[50,60],[70,80]],axis=1) #valid
```

In [386]:

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

Out[386]:

```
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11],
       [10, 20, 30]])
```

In [387]:

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

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-387-4f71b3d245ce> in <module>
----> 1 np.append(a,[[10,20,30]],axis=1) #invalid

<__array_function__ internals> in append(*args, **kwargs)

F:\Users\Gopi\anaconda3\lib\site-packages\numpy\lib\function_base.py in appen
d(arr, values, axis)
   4743         values = ravel(values)
   4744         axis = arr.ndim-1
-> 4745     return concatenate((arr, values), axis=axis)
   4746
   4747

<__array_function__ internals> in concatenate(*args, **kwargs)

ValueError: all the input array dimensions for the concatenation axis must ma
tch exactly, but along dimension 0, the array at index 0 has size 4 and the a
rray at index 1 has size 1
```

In [388]:

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

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-388-52259d5553e8> in <module>
----> 1 np.append(a,[[10],[20],[30]],axis=0) #invalid
```

```
<__array_function__ internals> in append(*args, **kwargs)

F:\Users\Gopi\anaconda3\lib\site-packages\numpy\lib\function_base.py in appen
d(arr, values, axis)
   4743         values = ravel(values)
   4744         axis = arr.ndim-1
-> 4745     return concatenate((arr, values), axis=axis)
   4746
   4747

<__array_function__ internals> in concatenate(*args, **kwargs)

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

In [389]:

np.append(a,[[10],[20],[30],[40]],axis=1) #valid

Out[389]:

```
array([[ 0,  1,  2, 10],
       [ 3,  4,  5, 20],
       [ 6,  7,  8, 30],
       [ 9, 10, 11, 40]])
```

In [390]:

np.append(a,[[10,20,30],[40,50,60]],axis=0) #valid

Out[390]:

```
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11],
       [10, 20, 30],
       [40, 50, 60]])
```

In [391]:

np.append(a,[[10,20],[30,40],[50,60],[70,80]],axis=1) #valid

Out[391]:

```
array([[ 0,  1,  2, 10, 20],
       [ 3,  4,  5, 30, 40],
       [ 6,  7,  8, 50, 60],
       [ 9, 10, 11, 70, 80]])
```

In [392]:

```
a = np.arange(12).reshape(4,3)
print(a)
b = np.append(a,[[10],[20],[30],[40]],axis=1)
print(b)
```

```
[[ 0  1  2]
 [ 3  4  5]
 [ 6  7  8]
 [ 9 10 11]]
[[ 0  1  2 10]
 [ 3  4  5 20]
 [ 6  7  8 30]
 [ 9 10 11 40]]
```

In [393]:

```
a = np.arange(12).reshape(4,3)
print(a)
b = np.append(a,[[10,50],[20,60],[30,70],[40,80]],axis=1)
print(b)
```

```
[[ 0  1  2]
 [ 3  4  5]
 [ 6  7  8]
 [ 9 10 11]]
[[ 0  1  2 10 50]
 [ 3  4  5 20 60]
 [ 6  7  8 30 70]
 [ 9 10 11 40 80]]
```

insert() vs append()

- By using insert() function, we can insert elements at our required index position.
- But by using append() function, we can add elements always at the end of ndarray.

Chapter-15
How to delete elements from ndarray

How to delete elements from ndarray

We can delete elements of ndarray by using delete() function.

delete(arr, obj, axis=None)

- obj can be int, array of ints or slice
- for multi-dimensional arrays we must have to specify the axis, other-wise the default axis=None will be considered. In this case first the array is flatten to the 1-D array and deletion will be performed

In [394]:

```
import numpy as np
help(np.delete)
```

```
Help on function delete in module numpy:

delete(arr, obj, axis=None)
    Return a new array with sub-arrays along an axis deleted. For a one
    dimensional array, this returns those entries not returned by
    `arr[obj]`.
```

1-D arrays

In [395]:

```
# To delete a single element of 1-D array at a specified index
a = np.arange(10,101,10)
b = np.delete(a,3) # to delete the element present at 3rd index
print(f"array a : {a}")
print(f"array b : {b}")
```

```
array a : [ 10  20  30  40  50  60  70  80  90 100]
array b : [ 10  20  30  50  60  70  80  90 100]
```

In [396]:

```
# To delete elements of 1-D array at a specified indices
a = np.arange(10,101,10)
b = np.delete(a,[0,4,6]) # to delete elements present at indices:0,4,6
print(f"array a : {a}")
print(f"array b : {b}")
```

```
array a : [ 10  20  30  40  50  60  70  80  90 100]
array b : [ 20  30  40  60  80  90 100]
```

In [397]:

```
# To delete elements of 1-D array from a specified range using numpy np.s_[]
a = np.arange(10,101,10)
b = np.delete(a,np.s_[2:6]) # to delete elements from 2nd index to 5th index
print(f"array a : {a}")
print(f"array b : {b}")
```

```
array a : [ 10  20  30  40  50  60  70  80  90 100]
array b : [ 10  20  70  80  90 100]
```

In [398]:

```
# To delete elements of 1-D array from a specified range using python range
# this is applicable only for 1-D arrays.
a = np.arange(10,101,10)
b = np.delete(a,range(2,6)) # to delete elements from 2nd index to 5th index
print(f"array a : {a}")
print(f"array b : {b}")
```

```
array a : [ 10  20  30  40  50  60  70  80  90 100]
array b : [ 10  20  70  80  90 100]
```

2-D arrays

- Here we have to provide axis.
- If we are not specifying access then array will be flatten(1-D) and then deletion will be happend.

In [399]:

```
# without providing the axis. defautl axis=None will be taken
a = np.arange(1,13).reshape(4,3)
b = np.delete(a,1)
```

```
print(f"array a : \n {a}")
print(f"array b : \n {b}")
```

```
array a :
 [[ 1  2  3]
 [ 4  5  6]
 [ 7  8  9]
 [10 11 12]]
array b :
 [ 1  3  4  5  6  7  8  9 10 11 12]
```

In [400]:

```
# axis=0. deleting the specifie row
a = np.arange(1,13).reshape(4,3)
b = np.delete(a,1,axis=0) # row at index 1 will be deleted
print(f"array a : \n {a}")
print(f"array b : \n {b}")
```

```
array a :
 [[ 1  2  3]
 [ 4  5  6]
 [ 7  8  9]
 [10 11 12]]
array b :
 [[ 1  2  3]
 [ 7  8  9]
 [10 11 12]]
```

In [401]:

```
# axis=0. deleting the specified rows
a = np.arange(1,13).reshape(4,3)
b = np.delete(a,[1,3],axis=0) # row at index 1 and index 3 will be deleted
print(f"array a : \n {a}")
print(f"array b : \n {b}")
```

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

```
array b :
 [[1 2 3]
 [7 8 9]]
```

In [402]:

```
# axis=0. deleting the specified range of rows
a = np.arange(1,13).reshape(4,3)
b = np.delete(a,np.s_[0:3],axis=0) # rows from index-0 to index-(3-1) will be deleted
print(f"array a : \n {a}")
print(f"array b : \n {b}")
```

```
array a :
 [[ 1  2  3]
 [ 4  5  6]
 [ 7  8  9]
 [10 11 12]]
array b :
 [[10 11 12]]
```

In [403]:

```
# axis=0. deleting the specified range of rows with step value
a = np.arange(1,13).reshape(4,3)
b =  np.delete(a,np.s_[::2],axis=0) # to delete every 2nd row (alternative row) from index-0
print(f"array a : \n {a}")
print(f"array b : \n {b}")
```

```
array a :
 [[ 1  2  3]
 [ 4  5  6]
 [ 7  8  9]
 [10 11 12]]
array b :
 [[ 4  5  6]
 [10 11 12]]
```

In [404]:

```
# axis=1. deleting the specifie column
a = np.arange(1,13).reshape(4,3)
b = np.delete(a,1,axis=1) # column at index 1 will be deleted
```

```
print(f"array a : \n {a}")
print(f"array b : \n {b}")
```

```
array a :
 [[ 1  2  3]
 [ 4  5  6]
 [ 7  8  9]
 [10 11 12]]
array b :
 [[ 1  3]
 [ 4  6]
 [ 7  9]
 [10 12]]
```

In [405]:

```
# axis=1. deleting the specified colunns
a = np.arange(1,13).reshape(4,3)
b = np.delete(a,[1,2],axis=1) # columns at index 1 and index 2 will be deleted
print(f"array a : \n {a}")
print(f"array b : \n {b}")
```

```
array a :
 [[ 1  2  3]
 [ 4  5  6]
 [ 7  8  9]
 [10 11 12]]
array b :
 [[ 1]
 [ 4]
 [ 7]
 [10]]
```

In [406]:

```
# axis=1. deleting the specified range of columns
a = np.arange(1,13).reshape(4,3)
b = np.delete(a,np.s_[0:2],axis=1) # rows from index-0 to index-(2-1) will be deleted
print(f"array a : \n {a}")
print(f"array b : \n {b}")
```

```
array a :
 [[ 1  2  3]
 [ 4  5  6]
```

```
 [ 7  8  9]
 [10 11 12]]
array b :
 [[ 3]
 [ 6]
 [ 9]
 [12]]
```

3-D arrays

In [407]:

```
# axis= None
a = np.arange(24).reshape(2,3,4)
b = np.delete(a,3) # flatten to 1-D array and element at 3rd index will be deleted
print(f"array a : \n {a}")
print(f"array b : \n {b}")
```

```
array a :
 [[[ 0  1  2  3]
  [ 4  5  6  7]
  [ 8  9 10 11]]

 [[12 13 14 15]
  [16 17 18 19]
  [20 21 22 23]]]
array b :
 [ 0  1  2  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23]
```

In [408]:

```
# axis=0. Delete a 2-D array at the specified index
a = np.arange(24).reshape(2,3,4)
b = np.delete(a,0,axis=0) # 2-D array at index 0 will be deleted
print(f"array a : \n {a}")
print(f"array b : \n {b}")
```

```
array a :
 [[[ 0  1  2  3]
  [ 4  5  6  7]
  [ 8  9 10 11]]

 [[12 13 14 15]
  [16 17 18 19]
```

```
  [20 21 22 23]]]
array b :
 [[[12 13 14 15]
  [16 17 18 19]
  [20 21 22 23]]]
```

In [409]:

```
# axis=1. Delete a row at the specified index from every 2-D array.
a = np.arange(24).reshape(2,3,4)
b = np.delete(a,0,axis=1) # Row at index 0 will be deleted from every 2-D array
print(f"array a : \n {a}")
print(f"array b : \n {b}")
```

```
array a :
 [[[ 0  1  2  3]
  [ 4  5  6  7]
  [ 8  9 10 11]]

 [[12 13 14 15]
  [16 17 18 19]
  [20 21 22 23]]]
array b :
 [[[ 4  5  6  7]
  [ 8  9 10 11]]

 [[16 17 18 19]
  [20 21 22 23]]]
```

In [410]:

```
# axis=1. Delete a column at the specified index from every 2-D array.
a = np.arange(24).reshape(2,3,4)
b = np.delete(a,1,axis=2) # column at index 1 will be deleted from every 2-D array
print(f"array a : \n {a}")
print(f"array b : \n {b}")
```

```
array a :
 [[[ 0  1  2  3]
  [ 4  5  6  7]
  [ 8  9 10 11]]
 [[12 13 14 15]
  [16 17 18 19]
  [20 21 22 23]]]
```

```
array b :
 [[[ 0  2  3]
  [ 4  6  7]
  [ 8 10 11]]

 [[12 14 15]
  [16 18 19]
  [20 22 23]]]
```

case study

Consider the following array

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

```
Delete last row and insert following row in that place:==> [70,80,90]
```

In [411]:

```
# Solution for the case study

# Step:1 ==> create the required ndarray
a = np.arange(12).reshape(4,3)

# Step:2 ==> Delete the last row.
# We have to mention the axis as axis=0 for rows.
# obj value we can  take as -1 , which represents the last row
b = np.delete(a,-1,axis=0)
print("Original Array")
print(f"array a : \n {a}")
print("After deletion the array ")
print(f"array b : \n {b}")

# Step:3 ==> we have to insert the row [70.80.90] at the end
# for this we can use append() function with axis=0
c = np.append(b,[[70,80,90]],axis=0)
print("After insertion the array ")
print(f"array c : \n {c}")
```

```
Original Array
array a :
```

```
 [[ 0  1  2]
 [ 3  4  5]
 [ 6  7  8]
 [ 9 10 11]]
After deletion the array
array b :
 [[0 1 2]
 [3 4 5]
 [6 7 8]]
After insertion the array
array c :
 [[ 0  1  2]
 [ 3  4  5]
 [ 6  7  8]
 [70 80 90]]
```

Summary

- **insert()** ==> Insert elements into an array at specified index.
- **append()** ==> Append elements at the end of an array.
- **delete()** ==> Delete elements from an array.

Chapter-16
Matrix multiplication by using dot() function

Matrix multiplication by using dot() function

- For a given two ndarrays a,b, if we perform a*b ==> element level multiplication will be happened
- To perform matrix level multiplication we have to use dot() function
- dot() function in available in numpy module
- dot() method is present in ndarray class also

In [412]:

```
# dot() function in numpy module
import numpy as np
help(np.dot)
```

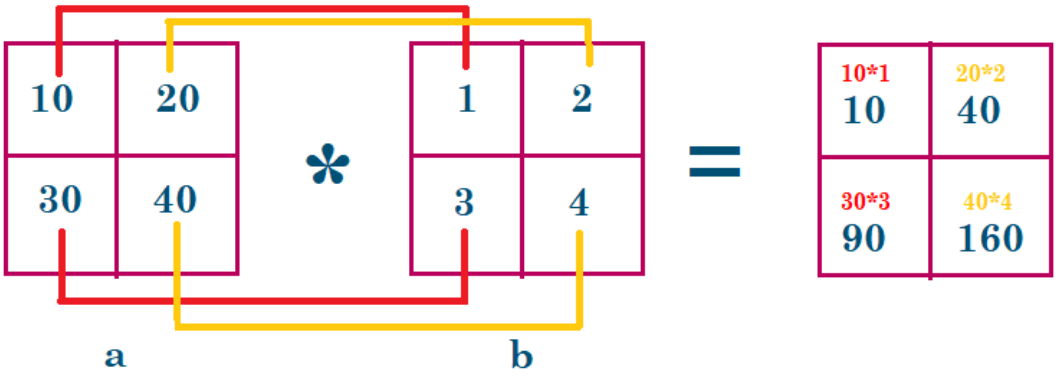
```
Help on function dot in module numpy:

dot(...)
    dot(a, b, out=None)
```

In [413]:

```
# dot() method in ndarray class
help(np.ndarray.dot)
```

```
Help on method_descriptor:

dot(...)
    a.dot(b, out=None)

    Dot product of two arrays.
```

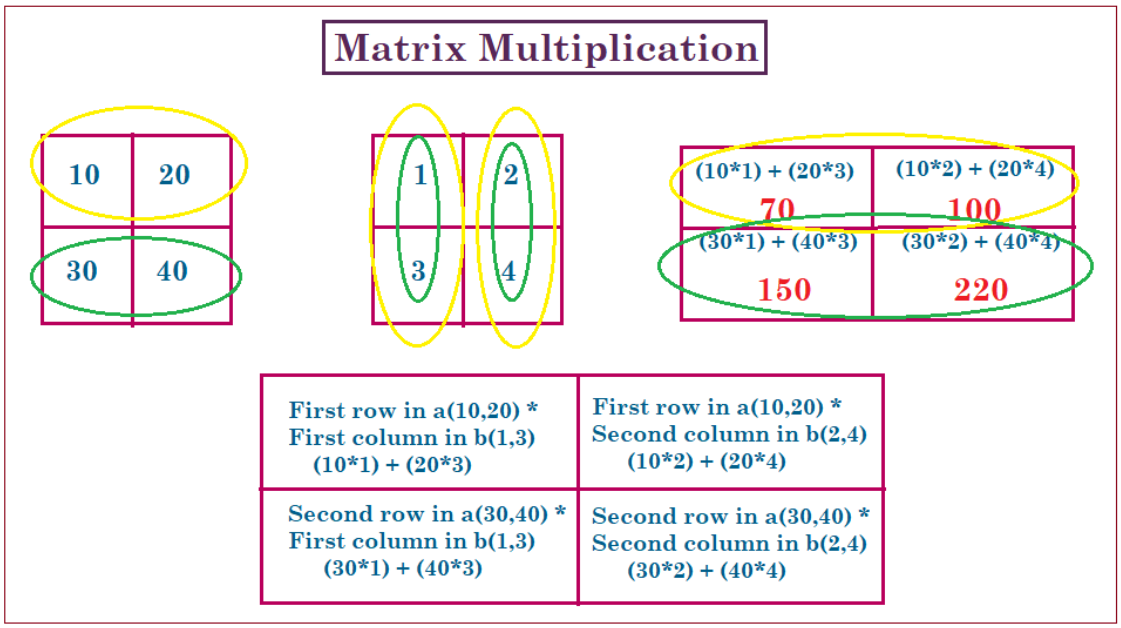
Element level Multiplication



In [414]:

```
# Element level multiplication
a = np.array([[10,20],[30,40]])
b = np.array([[1,2],[3,4]])
print(f"array a : \n {a}")
print(f"array b : \n {b}")
print("Element level multiplication ")
ele_multiplication = a* b
print(f"array b : \n {ele_multiplication}")
```

```
array a :
 [[10 20]
 [30 40]]
array b :
 [[1 2]
 [3 4]]
Element level multiplication
array b :
 [[ 10  40]
 [ 90 160]]
```



In [415]:

```
# matrix multiplication using dot() function in numpy module
a = np.array([[10,20],[30,40]])
b = np.array([[1,2],[3,4]])
dot_product = np.dot(a,b)
print(f"array a : \n {a}")
print(f"array b : \n {b}")
print(f"Matrix multiplication:\n {dot_product} ")
```

```
array a :
 [[10 20]
 [30 40]]
array b :
 [[1 2]
 [3 4]]
Matrix multiplication:
 [[ 70 100]
 [150 220]]
```

In [416]:

```
# matrix multiplication using dot() method in ndarray class
a = np.array([[10,20],[30,40]])
b = np.array([[1,2],[3,4]])
```

```
dot_product = a.dot(b)
print(f"array a : \n {a}")
print(f"array b : \n {b}")
print(f"Matrix multiplication:\n {dot_product} ")
```

```
array a :
 [[10 20]
 [30 40]]
array b :
 [[1 2]
 [3 4]]
Matrix multiplication:
 [[ 70 100]
 [150 220]]
```

Chapter-17
Importance of matrix class in numpy library

Importance of matrix class in numpy library

1-D array is called => Vector

2-D array is called => Matrix

- matrix class is specially designed class to create 2-D arrays.

In [417]:

```
import numpy as np
help(np.matrix)
```

```
Help on class matrix in module numpy:

class matrix(ndarray)
 |  matrix(data, dtype=None, copy=True)
 |
 |  matrix(data, dtype=None, copy=True)
 |
 |  .. note:: It is no longer recommended to use this class, even for linear
 |            algebra. Instead use regular arrays. The class may be removed
 |            in the future.
```

creating 2-D arrays

- By using matrix class
- By using ndarray class

class matrix(ndarray)

matrix(data, dtype=None, copy=True)

- data : array_like or string
- If `data` is a string, it is interpreted as a matrix with commas or spaces separating columns, and semicolons separating rows.

Parameters

1. data : array_like or string

- If `data` is a string, it is interpreted as a matrix with commas
- or spaces separating columns, and semicolons separating rows.

2. dtype : data-type

- Data-type of the output matrix.

3. copy : bool

- If `data` is already an `ndarray`, then this flag determines
- whether the data is copied (the default), or whether a view is constructed.

In [418]:

```
# Creating matrix object from string
# a = np.matrix('col1 col2 col3;col1 col2 col3')
# a = np.matrix('col1,col2,col3;col1,col2,col3')
a = np.matrix('10,20;30,40')
b = np.matrix('10 20;30 40')
print(f"type of a : type(a)")
print(f"type of b : type(b)")
print(f"Matrix object creation from string with comma : \n{a}")
print(f"Matrix object creation from string with space : \n{b}")
```

```
type of a : type(a)
type of b : type(b)
Matrix object creation from string with comma :
[[10 20]
 [30 40]]
Matrix object creation from string with space :
[[10 20]
 [30 40]]
```

In [419]:

```
#  Creating matrix object from nested list
a = np.matrix([[10,20],[30,40]])
a
```

Out[419]:

```
matrix([[10, 20],
        [30, 40]])
```

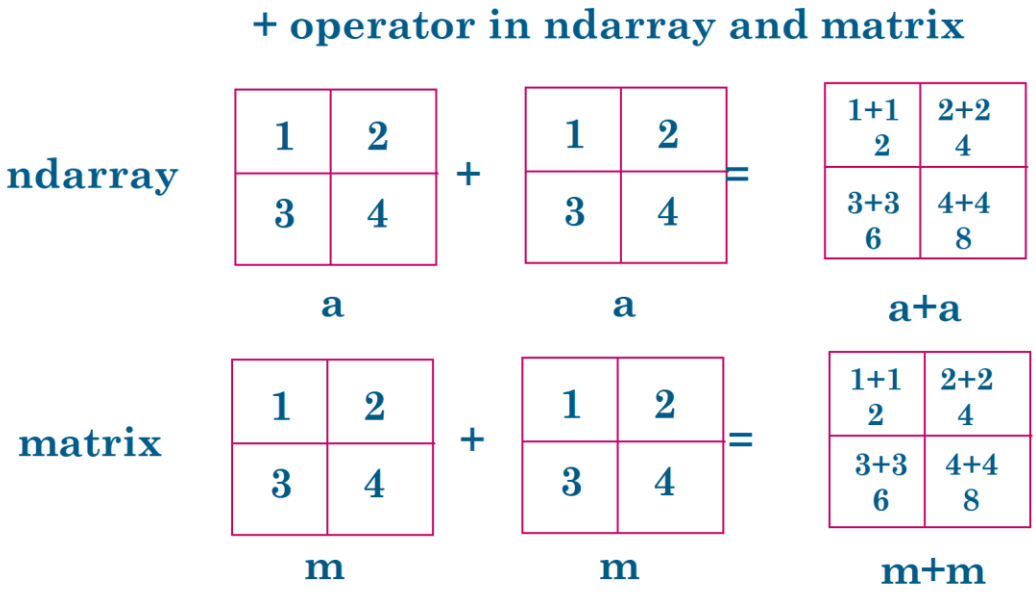
In [420]:

```
# create a matrix from ndarray
a = np.arange(6).reshape(3,2)
b = np.matrix(a)
print(f"type of a : type(a)")
print(f"type of b : type(b)")
print(f'ndarray :\n {a}')
print(f'matrix :\n {b}')
```

```
type of a : type(a)
type of b : type(b)
ndarray :
 [[0 1]
 [2 3]
 [4 5]]
matrix :
 [[0 1]
 [2 3]
 [4 5]]
```

+ operator in ndarray and matrix

- In case of both ndarray and matrix + operator behaves in the same way



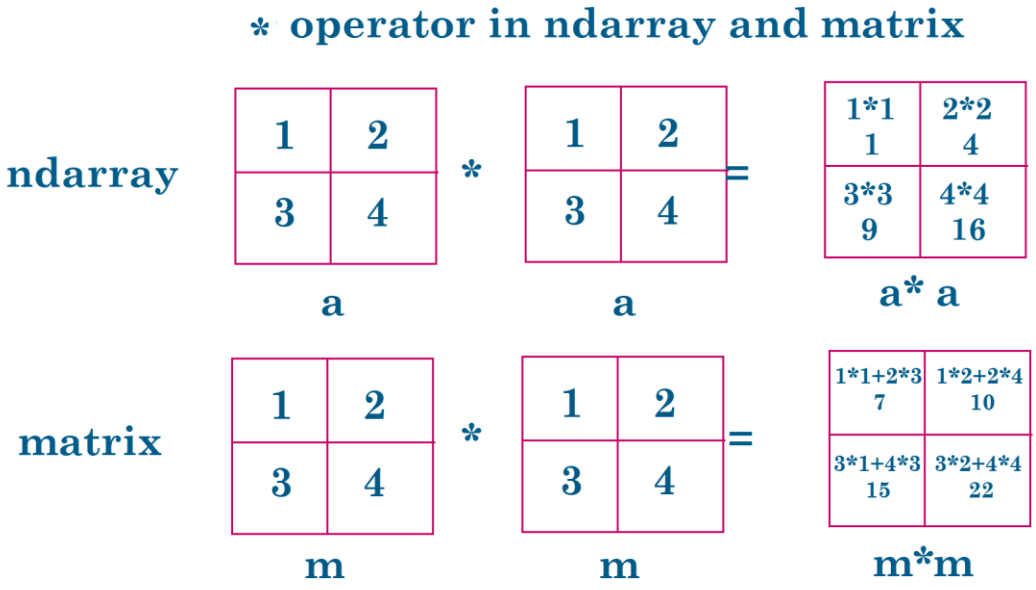
In [421]:

```
# + operator in ndarray and matrix
a = np.array([[1,2],[3,4]])
m = np.matrix([[1,2],[3,4]])

addition_a = a+a
addition_m = m+m

print(f'ndarray addition :\n {addition_a}')
print(f'matrix addition :\n {addition_m}')
```

```
ndarray addition :
 [[2 4]
 [6 8]]
matrix addition :
 [[2 4]
 [6 8]]
```

* operator in ndarray and matrix

- In case of ndarray * operator performs element level multiplication
- In case of matrix * operator performs matrix multiplication



In [422]:

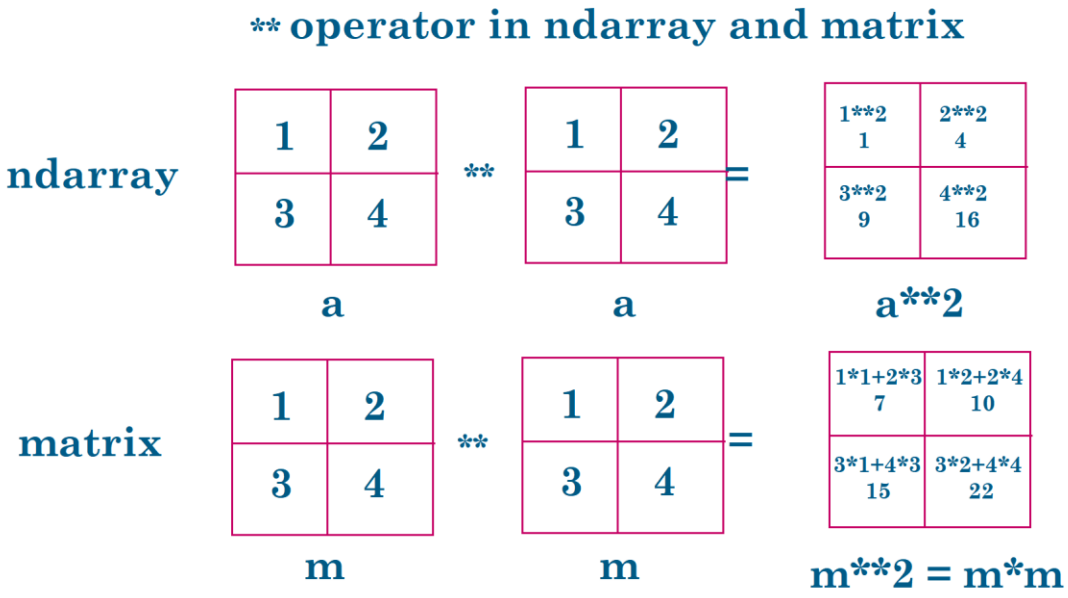
```
# * operator in ndarray and matrix
a = np.array([[1,2],[3,4]])
m = np.matrix([[1,2],[3,4]])

element_mul = a*a
matrix_mul = m*m

print(f'ndarray multiplication :\n {element_mul}')
print(f'matrix multiplication :\n {matrix_mul}')
```

```
ndarray multiplication :
 [[ 1  4]
 [ 9 16]]
matrix multiplication :
 [[ 7 10]
 [15 22]]
```

** operator in ndarray and

- In case of ndarray ** operator performs power operation at element level
- In case of matrix ** operator performs power operation at matrix level

m ** *2 ==> m* *m



In [423]:

```
# ** operator in ndarray and matrix
a = np.array([[1,2],[3,4]])
m = np.matrix([[1,2],[3,4]])

element_power = a**2
matrix_power = m**2

print(f'ndarray power :\n {element_power}')
print(f'matrix power :\n {matrix_power}')
```

```
ndarray power :
 [[ 1  4]
 [ 9 16]]
matrix power :
 [[ 7 10]
 [15 22]]
```

T in ndarray and matrix

- In case of both ndarray and matrix T behaves in the same way

In [424]:

```
# ** operator in ndarray and matrix
a = np.array([[1,2],[3,4]])
m = np.matrix([[1,2],[3,4]])

ndarray_T = a.T
matrix_T = m.T

print(f'ndarray transpose :\n {ndarray_T}')
print(f'matrix transpose :\n {matrix_T}')
```

```
ndarray transpose :
 [[1 3]
 [2 4]]
matrix transpose :
 [[1 3]
 [2 4]]
```

Conclusions

- matrix class is the child class of ndarray class. Hence all methods and properties of ndarray class are bydefault available to the matrix class.
- We can use `+, *, T, **` for matrix objects also.
- In the case of ndarray, *operator performs element level multiplication. But in case of matrix,* operator preforms matrix multiplication.
- In the case of ndarray, **operator performs power operation at element level. But in the case of matrix,** operator performs 'matrix' power.
- matrix class always meant for 2-D array only.
- It is no longer recommended to use.

Differences between ndarray and matrix

ndarray

- It can represent any n-dimension array.
- We can create from any array_like object but not from string.
- * operator meant for element mulitplication but not for dot product.
- ** operator meant for element level power operation
- It is the parent class
- It is the recommended to use

matrix

- It can represent only 2-dimension array.
- We can create from either array_like object or from string
- * operator meant for for dot product but not for element mulitplication.
- ** operator meant for for matrix power operation
- It is the child class
- It is not recommended to use and it is deprecated.

Chapter-18
Linear Algebra function from linalg module

Linear Algebra function from linalg module

numpy.linalg ==> contains functions to perform linear algebra operations

- **inv()** ==> to find inverse of a matrix
- **matrix_power()** ==> to find power of a matrix like A^n
- **det** ==> to find the determinant of a matrix
- **solve()** ==> to solve linear algebra equations

inv() --> to find inverse of a matrix

In [425]:

```
import numpy as np
help(np.linalg.inv)
```

```
Help on function inv in module numpy.linalg:

inv(a)
    Compute the (multiplicative) inverse of a matrix.
```

inverse of 2-D array(matrix)

To find Inverse of a Matrix

$$A = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \quad A^{-1} = \frac{1}{(ad-bc)} \begin{bmatrix} d & -b \\ -c & a \end{bmatrix}$$

$$A * A^{-1} = I$$

Matrix * Inverse Matrix = Identity Matrix

Example

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \quad A^{-1} = \frac{1}{(1*4+2*3)} \begin{bmatrix} 4 & -2 \\ -3 & 1 \end{bmatrix}$$

$$= \frac{1}{-2} \begin{bmatrix} 4 & -2 \\ -3 & 1 \end{bmatrix} = \begin{bmatrix} -2.0 & 1.0 \\ 1.5 & -0.5 \end{bmatrix}$$

In [426]:

```
# To Find inverse of a matrix
a = np.array([[1,2],[3,4]])
ainv = np.linalg.inv(a)
print(f"Original Matrix :: \n {a}")
print(f"Inverse of the Matrix :: \n {ainv}")
```

```
Original Matrix ::
 [[1 2]
 [3 4]]
Inverse of the Matrix ::
 [[-2.   1. ]
 [ 1.5 -0.5]]
```

How to check the inverse of the matrix is correct or not

- **dot(a,ainv) = dot(ainv,a) = eye(a,shape[0])**

In [427]:

```
# To check the inverse of the matrix correct or not
a = np.array([[1,2],[3,4]])
ainv = np.linalg.inv(a)
dot_produt = np.dot(a,ainv)
i = np.eye(2) # Identity matrix for the shape 2 ==> 2-D Arrray
print(f"Original Matrix :: \n {a}")
print(f"Inverse of the Matrix :: \n {ainv}")
print(f"Dot product of Matrix and Inverse of Matrix :: \n {dot_produt}")
print(f"Identity Matrix(2-D array using eye() function):: \n {i}")
```

```
Original Matrix ::
 [[1 2]
 [3 4]]
Inverse of the Matrix ::
 [[-2.   1. ]
 [ 1.5 -0.5]]
Dot product of Matrix and Inverse of Matrix ::
 [[1.00000000e+00 1.11022302e-16]
 [0.00000000e+00 1.00000000e+00]]
Identity Matrix(2-D array using eye() function)::
 [[1. 0.]
 [0. 1.]]
```

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \qquad A^{-1} = \begin{bmatrix} -2.0 & 1.0 \\ 1.5 & -0.5 \end{bmatrix}$$

$$A * A^{-1} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} * \begin{bmatrix} -2.0 & 1.0 \\ 1.5 & -0.5 \end{bmatrix}$$

1*-2 + 2*1.5 -2 + 3	1*1 + 2*-0.5 1 + -1
3*-2 + 4*1.5 -6 + 6	3*1 + 4*-0.5 3 + -2

1	0
0	1

np.allclose()

- **np.allclose()** ==> method is used to check the matrices are equal at element level

Note

- The results of floating point arithmetic varies from platform to platform

In [428]:

```
# help allclose() function
help(np.allclose)
```

```
Help on function allclose in module numpy:

allclose(a, b, rtol=1e-05, atol=1e-08, equal_nan=False)
    Returns True if two arrays are element-wise equal within a tolerance.
```

In [429]:

```
# demo for allclose() function
a = np.array([[1,2],[3,4]])
ainv = np.linalg.inv(a)
dot_produt = np.dot(a,ainv)
np.allclose(dot_produt,i)
```

Out[429]:

```
True
```

Note: We can find inverse only for square matrices, otherwise we will get error(LinAlgError)

In [430]:

```
a = np.arange(10).reshape(5,2)
np.linalg.inv(a)
```

```
---------------------------------------------------------------------------
LinAlgError                               Traceback (most recent call last)
<ipython-input-430-bb2ecf710117> in <module>
      1 a = np.arange(10).reshape(5,2)
----> 2 np.linalg.inv(a)

<__array_function__ internals> in inv(*args, **kwargs)

F:\Users\Gopi\anaconda3\lib\site-packages\numpy\linalg\linalg.py in inv(a)
    538     a, wrap = _makearray(a)
    539     _assert_stacked_2d(a)
--> 540     _assert_stacked_square(a)
    541     t, result_t = _commonType(a)
    542

F:\Users\Gopi\anaconda3\lib\site-packages\numpy\linalg\linalg.py in _assert_s
tacked_square(*arrays)
    201         m, n = a.shape[-2:]
    202         if m != n:
--> 203             raise LinAlgError('Last 2 dimensions of the array must be
square')
    204
    205 def _assert_finite(*arrays):

LinAlgError: Last 2 dimensions of the array must be square
```

inverse of 3-D array

- 3-D array is the collection of 2-D arrays
- Finding inverse of 3-D array means finding the inverse of every 2-D array

In [431]:

```
a = np.arange(8).reshape(2,2,2)
ainv = np.linalg.inv(a)
print(f"Original 3-D array :: \n {a}")
print(f"Inverse of 3-D array :: \n {ainv}")
```

```
Original 3-D array ::
 [[[0 1]
  [2 3]]

 [[4 5]
  [6 7]]]
Inverse of 3-D array ::
 [[[-1.5  0.5]
  [ 1.   0. ]]

 [[-3.5  2.5]
  [ 3.  -2. ]]]
```

matrix_power()

to find power of a matrix like A^n

matrix_power(a, n) ==> Raise a square matrix to the (integer) power `n`.

- if **n == 0** ==> Identity Matrix
- if **n > 0** ==> Normal Power operation
- if **n < 0** ==> First inverse and then power operation for absolute value of n `abs(n)`

In [432]:

```
import numpy as np
help(np.linalg.matrix_power)
```

```
Help on function matrix_power in module numpy.linalg:

matrix_power(a, n)
    Raise a square matrix to the (integer) power `n`.

    For positive integers `n`, the power is computed by repeated matrix
    squarings and matrix multiplications. If ``n == 0``, the identity matrix
    of the same shape as M is returned. If ``n < 0``, the inverse
    is computed and then raised to the ``abs(n)``.
```

In [433]:

```
# if n=0 ==> Identity Matrix
a = np.array([[1,2],[3,4]])
matrix_power = np.linalg.matrix_power(a,0)
```

```
print(f"Original 2-D array(Matrix) :: \n {a}")
print(f"Matrix Power of 2-D array :: \n {matrix_power}")
```

```
Original 2-D array(Matrix) ::
 [[1 2]
 [3 4]]
Matrix Power of 2-D array ::
 [[1 0]
 [0 1]]
```

In [434]:

```
# if n > 0 ==> Normal power operation
a = np.array([[1,2],[3,4]])
matrix_power = np.linalg.matrix_power(a,2)
print(f"Original 2-D array(Matrix) :: \n {a}")
print(f"Matrix Power of 2-D array :: \n {matrix_power}")
```

```
Original 2-D array(Matrix) ::
 [[1 2]
 [3 4]]
Matrix Power of 2-D array ::
 [[ 7 10]
 [15 22]]
```

In [435]:

```
# if n < 0 ==> First inverse opeartion and then power operation
a = np.array([[1,2],[3,4]])
matrix_power = np.linalg.matrix_power(a,-2)
print(f"Original 2-D array(Matrix) :: \n {a}")
print(f"Matrix Power of 2-D array :: \n {matrix_power}")
```

```
Original 2-D array(Matrix) ::
 [[1 2]
 [3 4]]
Matrix Power of 2-D array ::
 [[ 5.5  -2.5 ]
 [-3.75  1.75]]
```

In [436]:

```
# dot product of inverse matrix => dot(ainv) * dot(ainv) = result of n < 0 case
a = np.array([[1,2],[3,4]])
```

```
ainv = np.linalg.inv(a)
dot_product = np.dot(ainv,ainv)
print(f"Original 2-D array(Matrix) :: \n {a}")
print(f"Inverse of Matrix :: \n {ainv}")
print(f"Dot product of Inverse of Matrix :: \n {dot_product}")
```

```
Original 2-D array(Matrix) ::
 [[1 2]
 [3 4]]
Inverse of Matrix ::
 [[-2.   1. ]
 [ 1.5 -0.5]]
Dot product of Inverse of Matrix ::
 [[ 5.5  -2.5 ]
 [-3.75  1.75]]
```

In [437]:

```
# matrix power of inverse matrix and abs(n)
a = np.array([[1,2],[3,4]])
ainv = np.linalg.inv(a)
matrix_power = np.linalg.matrix_power(ainv,2)
print(f"Original 2-D array(Matrix) :: \n {a}")
print(f"Inverse of Matrix :: \n {ainv}")
print(f"Matrix power of Inverse of Matrix and abs(n) :: \n {matrix_power}")
```

```
Original 2-D array(Matrix) ::
 [[1 2]
 [3 4]]
Inverse of Matrix ::
 [[-2.   1. ]
 [ 1.5 -0.5]]
Matrix power of Inverse of Matrix and abs(n) ::
 [[ 5.5  -2.5 ]
 [-3.75  1.75]]
```

$$a^{-2} = (a^{-1})^2 = a^{-1} * a^{-1}$$

a^{-2} = np.linalg.matrix_power(a,-2)

$(a^{-1})^2$ = np.linalg.matrix_power(ainv,2)
ainv = np.linalg.inv(a)

$a^{-1} * a^{-1}$ = np.dot(ainv,ainv)
ainv = np.linalg.inv(a)

Note: We can find matrix_power only for square matrices, otherwise we will get error(LinAlgError)

In [438]:

```
a = np.arange(10).reshape(5,2)
np.linalg.matrix_power(a,2)
```

```
---------------------------------------------------------------------------
LinAlgError                               Traceback (most recent call last)
<ipython-input-438-99f952561699> in <module>
      1 a = np.arange(10).reshape(5,2)
----> 2 np.linalg.matrix_power(a,2)

<__array_function__ internals> in matrix_power(*args, **kwargs)

F:\Users\Gopi\anaconda3\lib\site-packages\numpy\linalg\linalg.py in matrix_po
wer(a, n)
    618     a = asanyarray(a)
    619     _assert_stacked_2d(a)
--> 620     _assert_stacked_square(a)
    621
    622     try:

F:\Users\Gopi\anaconda3\lib\site-packages\numpy\linalg\linalg.py in _assert_s
tacked_square(*arrays)
    201         m, n = a.shape[-2:]
    202         if m != n:
--> 203             raise LinAlgError('Last 2 dimensions of the array must be
```

```
square')
    204
    205 def _assert_finite(*arrays):

LinAlgError: Last 2 dimensions of the array must be square
```

det()

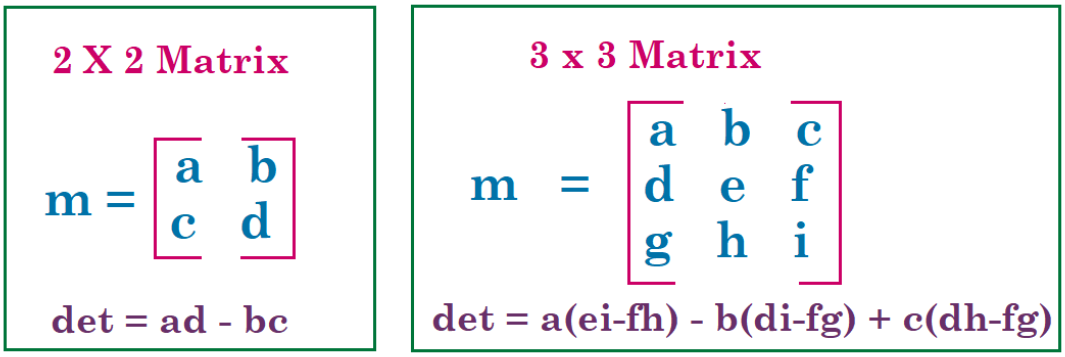
to find the determinant of a matrix

- determinant of the matrix = ad-bc

In [439]:

```
import numpy as np
help(np.linalg.det)
```

```
Help on function det in module numpy.linalg:

det(a)
    Compute the determinant of an array.
```



determinant of 2-D arrays

In [440]:

```
a = np.array([[1,2],[3,4]])
adet = np.linalg.det(a)
print(f"Original Matrix : \n {a}")
print(f"Determinant of  Matrix : \n {adet}")
```

```
Original Matrix :
 [[1 2]
 [3 4]]
Determinant of  Matrix :
 -2.0000000000000004
```

determinant of 3-D arrays

In [441]:

```
a = np.arange(9).reshape(3,3)
adet = np.linalg.det(a)
print(f"Original Matrix : \n {a}")
print(f"Determinant of  Matrix : \n {adet}")
```

```
Original Matrix :
 [[0 1 2]
 [3 4 5]
 [6 7 8]]
Determinant of  Matrix :
 0.0
```

Note: We can find determinant only for square matrices, otherwise we will get error(LinAlgError)

In [442]:

```
a = np.arange(10).reshape(5,2)
np.linalg.det(a)
```

```
---------------------------------------------------------------------------
LinAlgError                               Traceback (most recent call last)
<ipython-input-442-52a5a7a95a82> in <module>
      1 a = np.arange(10).reshape(5,2)
----> 2 np.linalg.det(a)

<__array_function__ internals> in det(*args, **kwargs)

F:\Users\Gopi\anaconda3\lib\site-packages\numpy\linalg\linalg.py in det(a)
   2153     a = asarray(a)
   2154     _assert_stacked_2d(a)
-> 2155     _assert_stacked_square(a)
   2156     t, result_t = _commonType(a)
   2157     signature = 'D->D' if isComplexType(t) else 'd->d'
```

```
F:\Users\Gopi\anaconda3\lib\site-packages\numpy\linalg\linalg.py in _assert_s
tacked_square(*arrays)
    201         m, n = a.shape[-2:]
    202         if m != n:
--> 203             raise LinAlgError('Last 2 dimensions of the array must be
square')
    204
    205 def _assert_finite(*arrays):

LinAlgError: Last 2 dimensions of the array must be square
```

solve()

to solve linear algebra equations

solve(a, b) ==> Solve a linear matrix equation, or system of linear scalar equations.

- a : (..., M, M) array_like ==> **Coefficient matrix.**
- b : {(..., M,), (..., M, K)}, array_like ==> **Ordinate** or **"dependent variable"** values.

2 variables

case study:

Problem:

- Boys and Girls are attending Durga sir's datascience class.
- For boys fee is $3 and for girls fee is $8.
- For a certain batch 2200 people attented and $10100 fee collected.
- How many boys and girls attended for that batch?

Assume that
x = number of boys
y = number of girls

Total Students = 2200 (x+y)
Total Fee collected = 10100(3x+8y)

x + y = 2200 --> 1
3x + 8y = 10100 --> 2

x = 2200 - y ---> 3
substitutin "x" value in equation 2 it will becomes

3(2200-y) + 8y = 10100 ---> 4

6600 - 3y +8y = 10100
5y = 10100 - 6600
y = 3500/5 ==> 700

Substituting y value in equation 1 we will get x value
x + 700 = 2200
x = 2200 - 700 ==> 1500

No. of Boys(x) = 1500
No. of Girls(y) = 700

x + y = 2200
3x + 8y = 10100

Coefficient Matrix

x	y
1	1
3	8

Ordinate/ Dependent

[2200,10100]

coef = np.array([[1,1],[3,8]])
ord = np.array([2200,10100])
np.linalg.solve(coef,ord)
=> array([1500. , 700.])

In [443]:

```
# solution
# x+y = 2200
# 3x+8y=10100

coef = np.array([[1,1],[3,8]])
dep = np.array([2200,10100])
result = np.linalg.solve(coef,dep)
print(f"Coefficient Matrix : \n {coef}")
print(f"Dependent Matrix : \n {dep}")
print(f"Solution array : {result}")
print(f"Type of result : {type(result)}")
```

```
Coefficient Matrix :
 [[1 1]
 [3 8]]
Dependent Matrix :
 [ 2200 10100]
Solution array : [1500.  700.]
Type of result : <class 'numpy.ndarray'>
```

3 variables

$$\begin{aligned} -4x + 7y - 2z &= 2 \\ x - 2y + z &= 3 \\ 2x - 3y + z &= -4 \end{aligned}$$

Coefficient Matrix

$$\begin{bmatrix} -4 & 7 & -2 \\ 1 & -2 & 1 \\ 2 & -3 & 1 \end{bmatrix}$$

Ordinate/ DependentVariable

[2 , 3, -4]

```
coef   = np.array([[-4,7,-2],[1,-2,1],[2,-3,1]])
dep    = np.array([2,3,-4])
result = np.linalg.solve(coef,dep)
```

In [444]:

```
# Finding the values of the 3 variables

# -4x+7y-2z = 2
# x-2y+z = 3
# 2x-3y+z = -4

coef = np.array([[-4,7,-2],[1,-2,1],[2,-3,1]])
dep =  np.array([2,3,-4])
result = np.linalg.solve(coef,dep)
print(f"Coefficient Matrix : \n {coef}")
print(f"Dependent Matrix : \n {dep}")
print(f"Solution array : {result}")
```

```
Coefficient Matrix :
 [[-4  7 -2]
 [ 1 -2  1]
 [ 2 -3  1]]
Dependent Matrix :
 [ 2  3 -4]
Solution array : [-13.  -6.   4.]
```

Chapter-19
IO operations with Numpy

I/O operations with Numpy

- We can save/write ndarray objects to a binary file for future purpose.
- Later point of time, when ever these objects are required, we can read from that binary file.
- **save()** ==> to save/write ndarry object to a file
- **load()** ==>to read ndarray object from a file

Syntax

- **save(file, arr, allow_pickle=True, fix_imports=True)** ==> Save an array to a binary file in NumPy `.npy` format.
- **load(file, mmap_mode=None, allow_pickle=False, fix_imports=True, encoding='ASCII')** ==> Load arrays or pickled objects from `.npy`, `.npz` or pickled files.

In [445]:

```
import numpy as np
help(np.save)
```

```
Help on function save in module numpy:

save(file, arr, allow_pickle=True, fix_imports=True)
    Save an array to a binary file in NumPy ``.npy`` format.
```

In [446]:

```
import numpy as np
help(np.load)
```

```
Help on function load in module numpy:

load(file, mmap_mode=None, allow_pickle=False, fix_imports=True, encoding='AS
CII')
    Load arrays or pickled objects from ``.npy``, ``.npz`` or pickled files.
```

save() and load() => single ndarray

In [447]:

```
# Saving ndarray object to a file and read back:(save_read_obj.py)
import numpy as np
a = np.array([[10,20,30],[40,50,60]]) #2-D array with shape:(2,3)

#save/serialize ndarray object to a file
np.save('out.npy',a)

#load/deserialize ndarray object from a file
out_array = np.load('out.npy')
print(out_array)
```

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

Note:

- The data will be stored in binary form
- File extension should be .npy, otherwise save() function itself will add that extension.
- By using save() function we can write only one obejct to the file. If we want to write multiple objects to a file then we should go for savez() function.

savez() and load() => multiple ndarrays

In [448]:

```
import numpy as np
help(np.savez)
```

```
Help on function savez in module numpy:

savez(file, *args, **kwds)
    Save several arrays into a single file in uncompressed ``.npz`` format.

    If arguments are passed in with no keywords, the corresponding variable
    names, in the ``.npz`` file, are 'arr_0', 'arr_1', etc. If keyword
    arguments are given, the corresponding variable names, in the ``.npz``
    file will match the keyword names.
```

In [449]:

```
# Saving mulitple ndarray objects to the binary file:
import numpy as np
a = np.array([[10,20,30],[40,50,60]]) #2-D array with shape:(2,3)
b = np.array([[70,80],[90,100]]) #2-D array with shape:(2,2)

#save/serialize ndarrays object to a file
np.savez('out.npz',a,b)

#reading ndarray objects from a file
npzfileobj = np.load('out.npz') #returns NpzFile object
print(f"Type of the npzfileobj : {type(npzfileobj)}")
print(npzfileobj.files)
print(npzfileobj['arr_0'])
print(npzfileobj['arr_1'])
```

```
Type of the npzfileobj : <class 'numpy.lib.npyio.NpzFile'>
['arr_0', 'arr_1']
[[10 20 30]
 [40 50 60]]
[[ 70  80]
 [ 90 100]]
```

In [450]:

```
# reading the file objets using for loop
import numpy as np
a = np.array([[10,20,30],[40,50,60]]) #2-D array with shape:(2,3)
b = np.array([[70,80],[90,100]]) #2-D array with shape:(2,2)

#save/serialize ndarrays object to a file
np.savez('out.npz',a,b)

#reading ndarray objects from a file
npzfileobj = np.load('out.npz') #returns NpzFile object ==> <class
'numpy.lib.npyio.NpzFile'>
print(type(npzfileobj))
print(npzfileobj.files)
for i in npzfileobj:
  print(f"Name of the file : {i}")
  print("Contents in the file :")
```

```
print(npzfileobj[i])
print("*"*80)
```

```
<class 'numpy.lib.npyio.NpzFile'>
['arr_0', 'arr_1']
Name of the file : arr_0
Contents in the file :
[[10 20 30]
 [40 50 60]]
********************************************************************************
Name of the file : arr_1
Contents in the file :
[[ 70  80]
 [ 90 100]]
********************************************************************************
```

Note:

- **np.save()** ==>Save an array to a binary file in .npy format
- **np.savez(** ==> Save several arrays into a single file in .npz format but in uncompressed form.
- **np.savez_compressed()** ==>Save several arrays into a single file in .npz format but in compressed form.
- **np.load()** ==>To load/read arrays from .npy or .npz files.

savez_compressed() and load() => ndarray compressed form

In [451]:

```
import numpy as np
help(np.savez_compressed)
```

```
Help on function savez_compressed in module numpy:

savez_compressed(file, *args, **kwds)
    Save several arrays into a single file in compressed ``.npz`` format.
```

In [452]:

```
# compressed form

import numpy as np
a = np.array([[10,20,30],[40,50,60]]) #2-D array with shape:(2,3)
```

```
b = np.array([[70,80],[90,100]]) #2-D array with shape:(2,2)

#save/serialize ndarrays object to a file
np.savez_compressed('out_compressed.npz',a,b)

#reading ndarray objects from a file
npzfileobj = np.load('out_compressed.npz') #returns NpzFile object
#print(type(npzfileobj))
print(npzfileobj.files)
print(npzfileobj['arr_0'])
print(npzfileobj['arr_1'])
```

```
['arr_0', 'arr_1']
[[10 20 30]
 [40 50 60]]
[[ 70  80]
 [ 90 100]]
```

```
In [453]:

# Analysys
%ls out.npz  out_compressed.npz
```

```
 Volume in drive D is BigData
 Volume Serial Number is 8E56-9F3B

 Directory of D:\Youtube_Videos\DurgaSoft\DataScience\JupyterNotebooks_Numpy\
Chapter_wise_Notes


 Directory of D:\Youtube_Videos\DurgaSoft\DataScience\JupyterNotebooks_Numpy\
Chapter_wise_Notes

15-08-2021  01:55               546 out.npz
15-08-2021  01:56               419 out_compressed.npz
               2 File(s)            965 bytes
               0 Dir(s)  85,327,192,064 bytes free
```

We can save object in compressed form, then what is the need of uncompressed form?

- **compressed form** ==>memory will be saved, but performance down.
- **uncompressed form** ==>memory won't be saved, but performance wise good.

Note:

- if we are using **save()** function the file extension: **npy**
- if we are using **savez() or savez_compressed()** functions the file extension: **npz**

savetxt() and loadtxt() => ndarray object to text file

- To save ndarray object to a text file we will use **savetxt()** function
- To read ndarray object from a text file we will use **loadtxt()** function

In [454]:

```
import numpy as np
help(np.savetxt)
```

```
Help on function savetxt in module numpy:

savetxt(fname, X, fmt='%.18e', delimiter=' ', newline='\n', header='', footer
='', comments='# ', encoding=None)
    Save an array to a text file.
```

In [455]:

```
import numpy as np
help(np.loadtxt)
```

```
Help on function loadtxt in module numpy:

loadtxt(fname, dtype=<class 'float'>, comments='#', delimiter=None, converter
s=None, skiprows=0, usecols=None, unpack=False, ndmin=0, encoding='bytes', ma
x_rows=None, *, like=None)
    Load data from a text file.

    Each row in the text file must have the same number of values.
```

In [456]:

```
import numpy as np
a = np.array([[10,20,30],[40,50,60]]) #2-D array with shape:(2,3)

#save/serialize ndarrays object to a file
np.savetxt('out.txt',a,fmt='%.1f')
```

```
#reading ndarray objects from a file and default dtype is float
out_array1 = np.loadtxt('out.txt')
print("Output array in default format : float")
print(out_array1)

#reading ndarray objects from a file and default dtype is int
print("Output array in int format")
out_array2 = np.loadtxt('out.txt',dtype=int)
print(out_array2)
```

```
Output array in default format : float
[[10. 20. 30.]
 [40. 50. 60.]]
Output array in int format
[[10 20 30]
 [40 50 60]]
```

In [457]:

```
## save ndarray object(str) into a text file
import numpy as np
a1 = np.array([['Sunny',1000],['Bunny',2000],['Chinny',3000],['Pinny',4000]])

#save this ndarray to a text file
np.savetxt('out.txt',a1) # by default fmt='%.18e'. It will leads to error
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
F:\Users\Gopi\anaconda3\lib\site-packages\numpy\lib\npyio.py in savetxt(fname
, X, fmt, delimiter, newline, header, footer, comments, encoding)
   1432                 try:
-> 1433                     v = format % tuple(row) + newline
   1434                 except TypeError as e:

TypeError: must be real number, not numpy.str_

The above exception was the direct cause of the following exception:

TypeError                                 Traceback (most recent call last)
<ipython-input-457-bf45be6c6703> in <module>
      4
      5 #save this ndarray to a text file
```

```
----> 6 np.savetxt('out.txt',a1) # by default fmt='%.18e'. It will leads to e
rror

<__array_function__ internals> in savetxt(*args, **kwargs)

F:\Users\Gopi\anaconda3\lib\site-packages\numpy\lib\npyio.py in savetxt(fname
, X, fmt, delimiter, newline, header, footer, comments, encoding)
   1433                     v = format % tuple(row) + newline
   1434                 except TypeError as e:
-> 1435                     raise TypeError("Mismatch between array dtype ('%
s') and "
   1436                                     "format specifier ('%s')"
   1437                                     % (str(X.dtype), format)) from e

TypeError: Mismatch between array dtype ('<U11') and format specifier ('%.18e
%.18e')
```

savetxt() conclusions

- By using savetxt() we can store ndarray of type 1-D and 2-D only
- If we use 3-D array to store into a file then it will give error

In [458]:

```
import numpy as np
a = np.arange(24).reshape(2,3,4)
np.savetxt('output.txt',a)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-458-24e4ed2480fe> in <module>
      1 import numpy as np
      2 a = np.arange(24).reshape(2,3,4)
----> 3 np.savetxt('output.txt',a)

<__array_function__ internals> in savetxt(*args, **kwargs)

F:\Users\Gopi\anaconda3\lib\site-packages\numpy\lib\npyio.py in savetxt(fname
, X, fmt, delimiter, newline, header, footer, comments, encoding)
   1378         # Handle 1-dimensional arrays
   1379         if X.ndim == 0 or X.ndim > 2:
-> 1380             raise ValueError(
   1381                 "Expected 1D or 2D array, got %dD array instead" % X.
ndim)
```

```
   1382         elif X.ndim == 1:

ValueError: Expected 1D or 2D array, got 3D array instead
```

In [459]:

```
# to store str ndarray into a file we must specify the fmt parameter as str
import numpy as np
a1 = np.array([['Sunny',1000],['Bunny',2000],['Chinny',3000],['Pinny',4000]])

#save this ndarray to a text file
np.savetxt('out.txt',a1,fmt='%s %s')

#reading ndarray from the text file
a2 = np.loadtxt('out.txt',dtype='str')
print(f"Type of a2(fetching the data from text file) : {type(a2)}")
print(f"The a2 value after fetching the data from text file : \n {a2}")
```

```
Type of a2(fetching the data from text file) : <class 'numpy.ndarray'>
The a2 value after fetching the data from text file :
 [['Sunny' '1000']
 ['Bunny' '2000']
 ['Chinny' '3000']
 ['Pinny' '4000']]
```

Creating ndarray object by reading a file

In [460]:

```
# out.txt:
# -------
# Sunny 1000
# Bunny 2000
# Chinny 3000
# Pinny 4000
# Zinny 5000
# Vinny 6000
# Minny 7000
# Tinny 8000
# creating ndarray from text file data:
import numpy as np
#reading ndarray from the text file
```

```
a2 = np.loadtxt('out.txt',dtype='str')
print(f"Type of a2 : {type(a2)}")
print(a2)
```

```
Type of a2 : <class 'numpy.ndarray'>
[['Sunny' '1000']
 ['Bunny' '2000']
 ['Chinny' '3000']
 ['Pinny' '4000']]
```

Writing ndarray objects to the csv file

- **csv** - Comma separated Values

```
In [461]:

import numpy as np

a1 = np.array([[10,20,30],[40,50,60]])

#save/serialize to a csv file
np.savetxt('out.csv',a1,delimiter=',')

#reading ndarray object from a csv file
a2 = np.loadtxt('out.csv',delimiter=',')
print(a2)
```

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

Summary:

- Save **one ndarray** object to the **binary file(save() and load())**
- Save **multiple ndarray** objects to the **binary file in uncompressed form(savez() and load())**
- Save **multiple ndarray** objects to the binary file in **compressed form(savez_compressed() and load())**
- Save **ndarry object** to the **text file (savetxt() and loadtxt())**
- Save **ndarry object** to the **csv file (savetxt() and loadtxt() with delimiter=',')**

Chapter-20
Basic Statistics with Numpy

Basic Statistics with Numpy

In Datascience domain, we required to collect,store and analyze huge amount of data. From this data we may required to find some basic statistics like

- Minimum value
- Maximum value
- Average Value
- Sum of all values
- Mean value
- Median value
- Variance
- Standard deviation etc

Minimum value

- **np.min(a)**
- **np.amin(a)**
- **a.min()**

In [462]:

```
import numpy as np
help(np.min)
```

```
Help on function amin in module numpy:

amin(a, axis=None, out=None, keepdims=<no value>, initial=<no value>, where=<
no value>)
    Return the minimum of an array or minimum along an axis.
```

1-D array

In [463]:

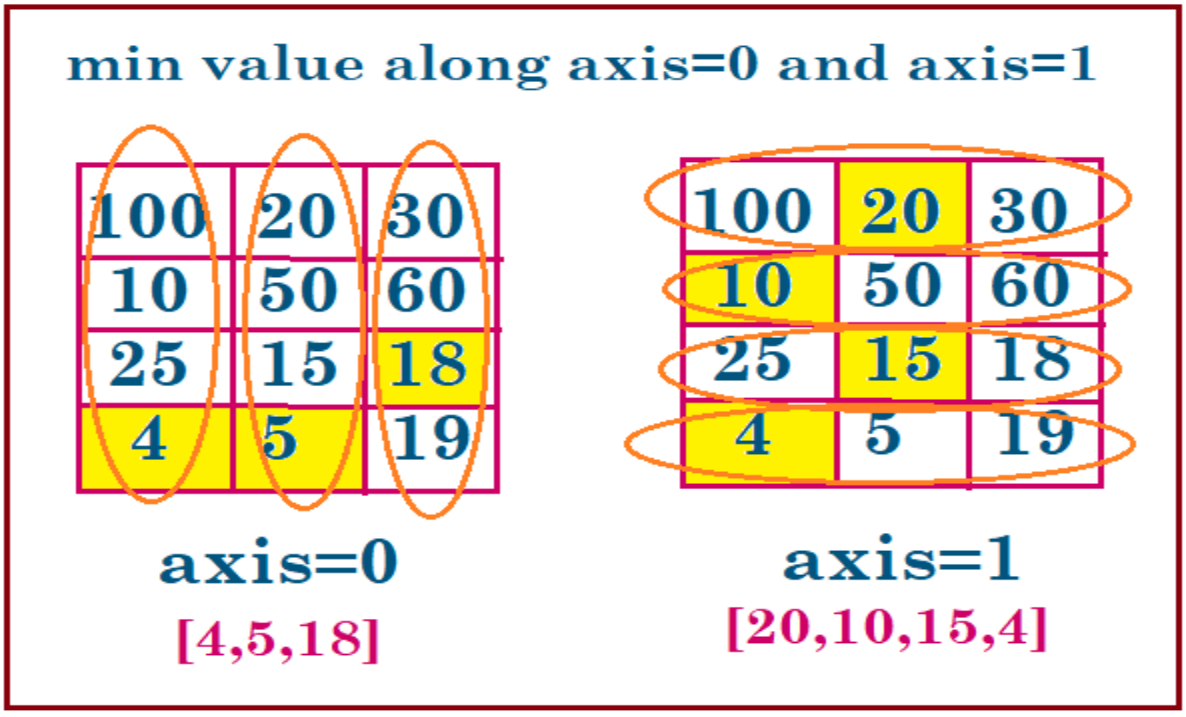
```
a = np.array([10,5,20,3,25])
print(f"1-D array : {a}")
```

```
print(f"np.min(a) value : {np.min(a)}")
print(f"np.amin(a) value : {np.amin(a)}")
print(f"a.min() value : {a.min()}")
```

```
1-D array : [10  5 20  3 25]
np.min(a) value : 3
np.amin(a) value : 3
a.min() value : 3
```

2-D array

- **axis=None(default)** - The array is flattened to 1-D array and find the the min value
- **axis=0** - minimum row and that row contains 3 element
- **axis=1** - minimum column and that column contains 4 elements
- **axis=0** - minimum row and that is by considering all the columns, in that min row value
- **axis=1** - minimum column and that is by considering all rows, in that min column value



In [464]:

```
import numpy as np
a = np.array([[100,20,30],[10,50,60],[25,15,18],[4,5,19]])
print(f"array a : \n {a}")
print(f"Minimum value along axis=None : {np.min(a)}")
print(f"Minimum value along axis-0 : {np.min(a,axis=0)}")
print(f"Minimum value along axis-1 : {np.min(a,axis=1)}")
```

```
array a :
 [[100  20  30]
 [ 10  50  60]
 [ 25  15  18]
 [  4   5  19]]
Minimum value along axis=None : 4
Minimum value along axis-0 : [ 4  5 18]
Minimum value along axis-1 : [20 10 15  4]
```

In [465]:

```
import numpy as np
a = np.arange(24).reshape(6,4)
print(f"array a : \n {a}")
print(f"Minimum value along axis=None : {np.min(a)}")
print(f"Minimum value along axis-0 : {np.min(a,axis=0)}")
print(f"Minimum value along axis-1 : {np.min(a,axis=1)}")
```

```
array a :
 [[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]
 [12 13 14 15]
 [16 17 18 19]
 [20 21 22 23]]
Minimum value along axis=None : 0
Minimum value along axis-0 : [0 1 2 3]
Minimum value along axis-1 : [ 0  4  8 12 16 20]
```

In [466]:

```
import numpy as np
a = np.arange(24)
np.random.shuffle(a)
a = a.reshape(6,4)
```

```
print(f"array a : \n {a}")
print(f"Minimum value along axis=None : {np.min(a)}")
print(f"Minimum value along axis-0 : {np.min(a,axis=0)}")
print(f"Minimum value along axis-1 : {np.min(a,axis=1)}")
```

```
array a :
 [[20  5  4 21]
 [ 1 10  6 14]
 [ 0 11 17 13]
 [ 3  2 22 23]
 [ 8  7 19 18]
 [ 9 12 15 16]]
Minimum value along axis=None : 0
Minimum value along axis-0 : [ 0  2  4 13]
Minimum value along axis-1 : [4 1 0 2 7 9]
```

Maximum value

- **np.max(a)**
- **np.amax(a)**
- **a.max()**

In [467]:

```
import numpy as np
help(np.max)
```

```
Help on function amax in module numpy:

amax(a, axis=None, out=None, keepdims=<no value>, initial=<no value>, where=<
no value>)
    Return the maximum of an array or maximum along an axis.
```

1-D array

In [468]:

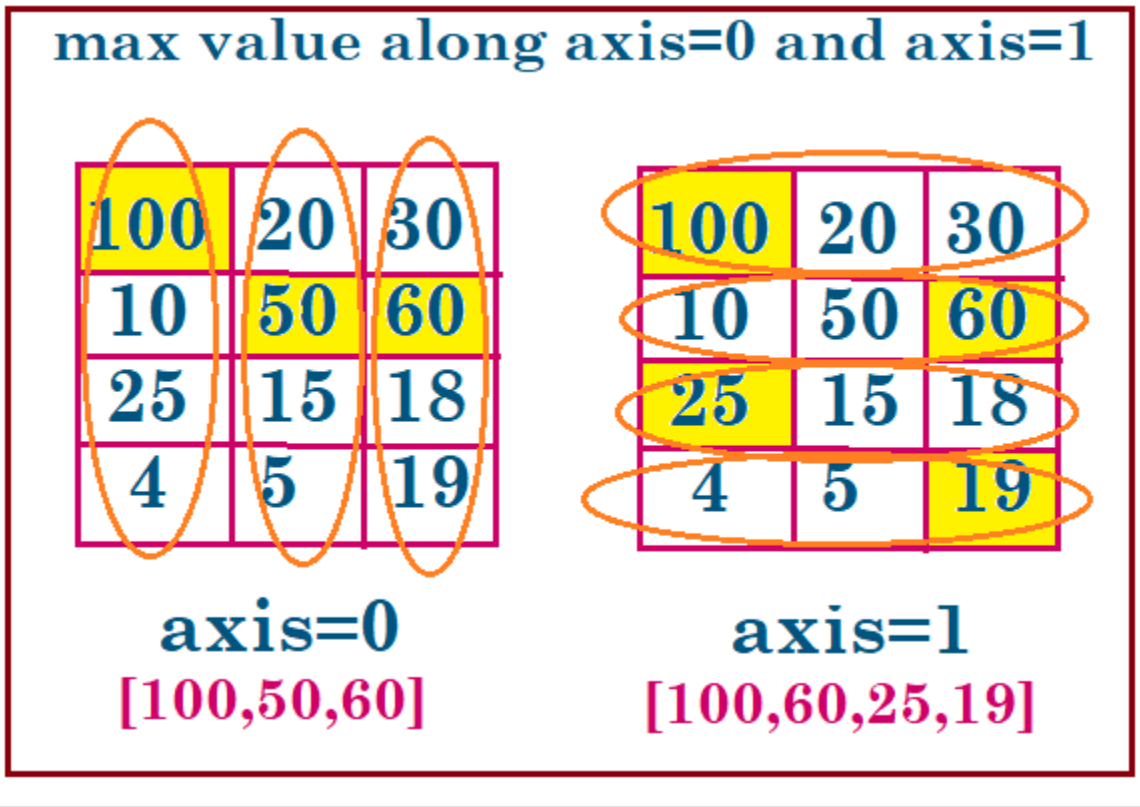
```
a = np.array([10,5,20,3,25])
print(f"1-D array : {a}")
print(f"np.max(a) value : {np.max(a)}")
print(f"np.amax(a) value : {np.amax(a)}")
print(f"a.max() value : {a.max()}")
```

```
1-D array : [10  5 20  3 25]
np.max(a) value : 25
```

```
np.amax(a) value : 25
a.max() value : 25
```

2-D array

- **axis=None(default)** - The array is flattened to 1-D array and find the the max value
- **axis=0** - maximum row and that row contains 3 element
- **axis=1** - maximum column and that column contains 4 elements
- **axis=0** - maximum row and that is by considering all the columns, in that max row value
- **axis=1** - maximum column and that is by considering all rows, in that max column value



In [469]:

```
import numpy as np
a = np.array([[100,20,30],[10,50,60],[25,15,18],[4,5,19]])
print(f"array a : \n {a}")
```

```
print(f"Maximum value along axis=None : {np.max(a)}")
print(f"Maximum value along axis-0 : {np.max(a,axis=0)}")
print(f"Maximum value along axis-1 : {np.max(a,axis=1)}")
```

```
array a :
 [[100  20  30]
 [ 10  50  60]
 [ 25  15  18]
 [  4   5  19]]
Maximum value along axis=None : 100
Maximum value along axis-0 : [100  50  60]
Maximum value along axis-1 : [100  60  25  19]
```

sum of the elements

- **np.sum()**
- **a.sum()**

In [470]:

```
import numpy as np
help(np.sum)
```

```
Help on function sum in module numpy:

sum(a, axis=None, dtype=None, out=None, keepdims=<no value>, initial=<no valu
e>, where=<no value>)
    Sum of array elements over a given axis.
```

1-D array

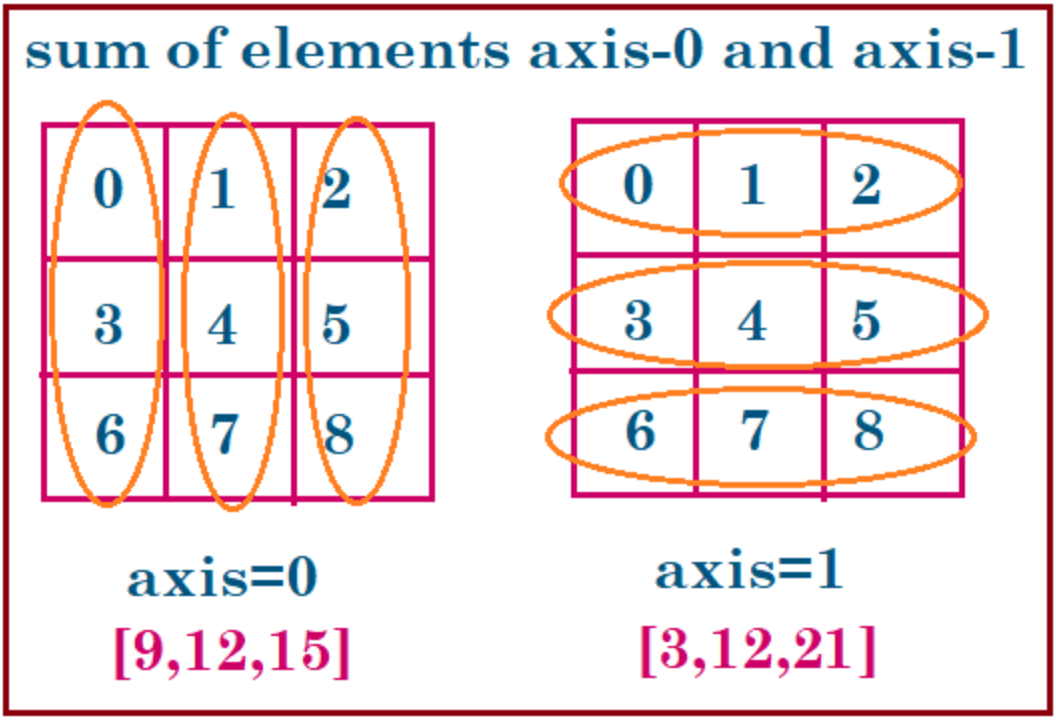
In [471]:

```
# sum of elements of 1-D array
a = np.arange(4)
print(f"The array a : {a}")
print(f"sum of elements using np.sum(a) :: {np.sum(a)}")
print(f"sum of elements using a.sum() :: {a.sum()}")
```

```
The array a : [0 1 2 3]
sum of elements using np.sum(a) :: 6
sum of elements using a.sum() :: 6
```

2-D array

- **axis=None(default)** - The array is flattened to 1-D array and sum is calculated
- **axis=0** - all rows and sum of each column
- **axis=1** - all columns and sum of each row




```
In [472]:

a = np.arange(9).reshape(3,3)
print(f"array a : \n {a}")
print(f"Sum along axis=None : {np.sum(a)}")
print(f"Sum along axis-0 : {np.sum(a,axis=0)}")
print(f"Sum along axis-1 : {np.sum(a,axis=1)}")
```

```
array a :
 [[0 1 2]
 [3 4 5]
 [6 7 8]]
Sum along axis=None : 36
Sum along axis-0 : [ 9 12 15]
Sum along axis-1 : [ 3 12 21]
```

Mean value

- **np.mean(a)**
- **a.mean()**

- Mean is the sum of elements along the specified axis divided by number of elements.

In [473]:

```
import numpy as np
help(np.mean)
```

```
Help on function mean in module numpy:

mean(a, axis=None, dtype=None, out=None, keepdims=<no value>, *, where=<no va
lue>)
    Compute the arithmetic mean along the specified axis.
```

1-D array

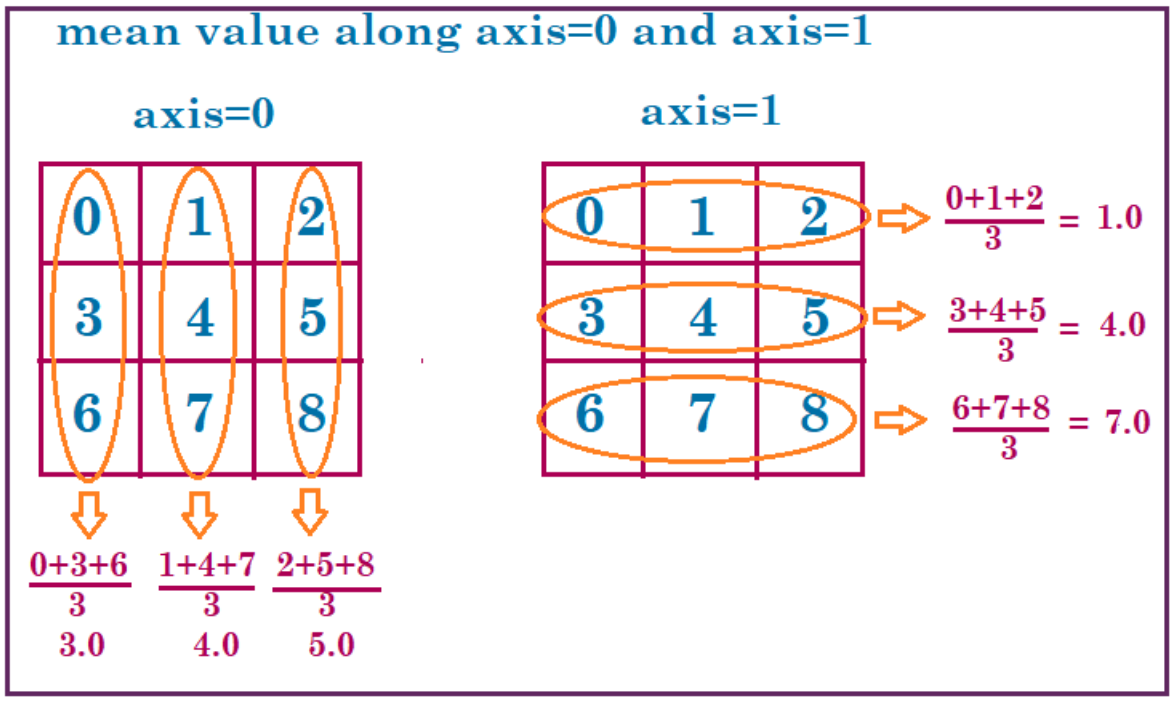
In [474]:

```
a = np.arange(5)
print(f"1-D array : {a}")
print(f"np.mean(a) value : {np.mean(a)}")
print(f"a.mean() value : {a.mean()}")
```

```
1-D array : [0 1 2 3 4]
np.mean(a) value : 2.0
a.mean() value : 2.0
```

2-D array

- **axis=None(default)** - The array is flattened to 1-D array and find the mean(average) value
- **axis=0** - rows. Consider columns with all rows and find the average
- **axis=1** - columns. Consider rows with all columns and find the average



In [475]:

```
# 2-D array mean
a = np.arange(9).reshape(3,3)
print(f"The original 2-D array : \n {a}")
print(f"Mean of the 2-D array along axis=None : {np.mean(a)}")
print(f"Mean of the 2-D array along axis=0 : {np.mean(a,axis=0)}")
print(f"Mean of the 2-D array along axis=1 : {np.mean(a,axis=1)}")
```

```
The original 2-D array :
 [[0 1 2]
 [3 4 5]
 [6 7 8]]
Mean of the 2-D array along axis=None : 4.0
Mean of the 2-D array along axis=0 : [3. 4. 5.]
Mean of the 2-D array along axis=1 : [1. 4. 7.]
```

Median value

np.median(a)

- Median means middle element of the array **(sorted form)**
- If the array contains **even number of elements**, then the **median** is the **middle element value**
- If the array contains **odd number of elements**, then the **median** is the **average of 2 middle element values**

In [476]:

```
import numpy as np
help(np.median)
```

```
Help on function median in module numpy:

median(a, axis=None, out=None, overwrite_input=False, keepdims=False)
    Compute the median along the specified axis.

    Returns the median of the array elements.
```

1-D array

In [477]:

```
a = np.array([10,20,30,40])
b = np.array([10,20,30,40,50])
print(f"The array with even number of elements : {a}")
print(f"Median of the array with even number of elements : {np.median(a)}")
print()
print(f"The array with odd number of elements : {b}")
print(f"Median of the array with odd number of elements : {np.median(b)}")
```

```
The array with even number of elements : [10 20 30 40]
Median of the array with even number of elements : 25.0

The array with odd number of elements : [10 20 30 40 50]
Median of the array with odd number of elements : 30.0
```

In [478]:

```
# unsorted array(even no of elements) will be converted to sorted array and then
#median is calculated
a = np.array([80,20,60,40])
print(f"The array with even number of elements(unsorted) : {a}")
print("*"*100)
print("This step is calculated internally ")
print(f"sorted form of given array : {np.sort(a)}")
print("*"*100)
print(f"Median of the array with even number of elements : {np.median(a)}")
```

```
The array with even number of elements(unsorted) : [80 20 60 40]
****************************************************************************************************
This step is calculated internally
sorted form of given array : [20 40 60 80]
****************************************************************************************************
Median of the array with even number of elements : 50.0
```

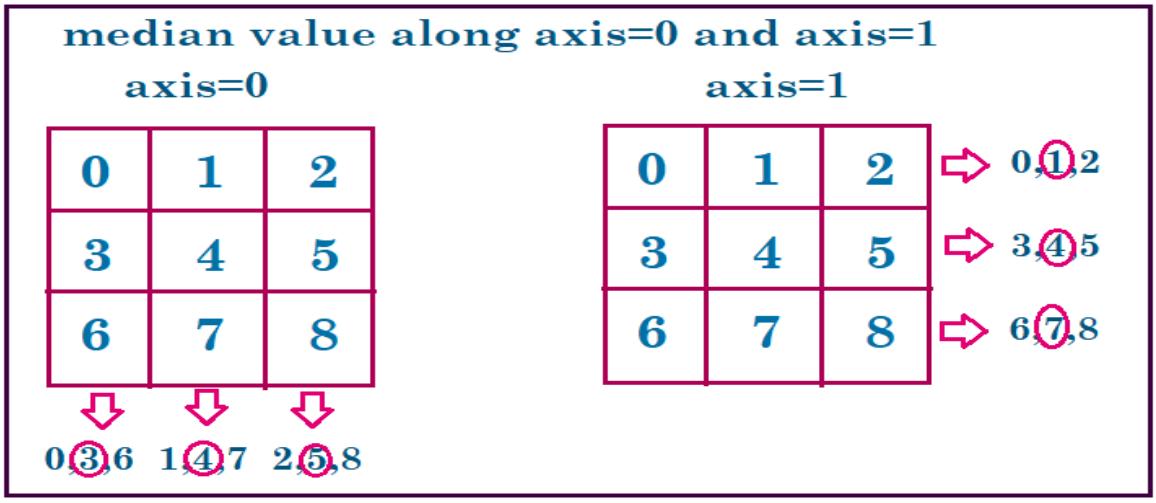
In [479]:

```
# unsorted array(odd no of elements) will be converted to sorted array and then
#median is calculated
a = np.array([80,20,60,40,100,140,120])
print(f"The array with even number of elements(unsorted) : {a}")
print("*"*100)
print("This step is calculated internally ")
print(f"sorted form of given array : {np.sort(a)}")
print("*"*100)
print(f"Median of the array with even number of elements : {np.median(a)}")
```

```
The array with even number of elements(unsorted) : [ 80  20  60  40 100 140 120]
****************************************************************************************************
This step is calculated internally
sorted form of given array : [ 20  40  60  80 100 120 140]
****************************************************************************************************
Median of the array with even number of elements : 80.0
```

2-D array

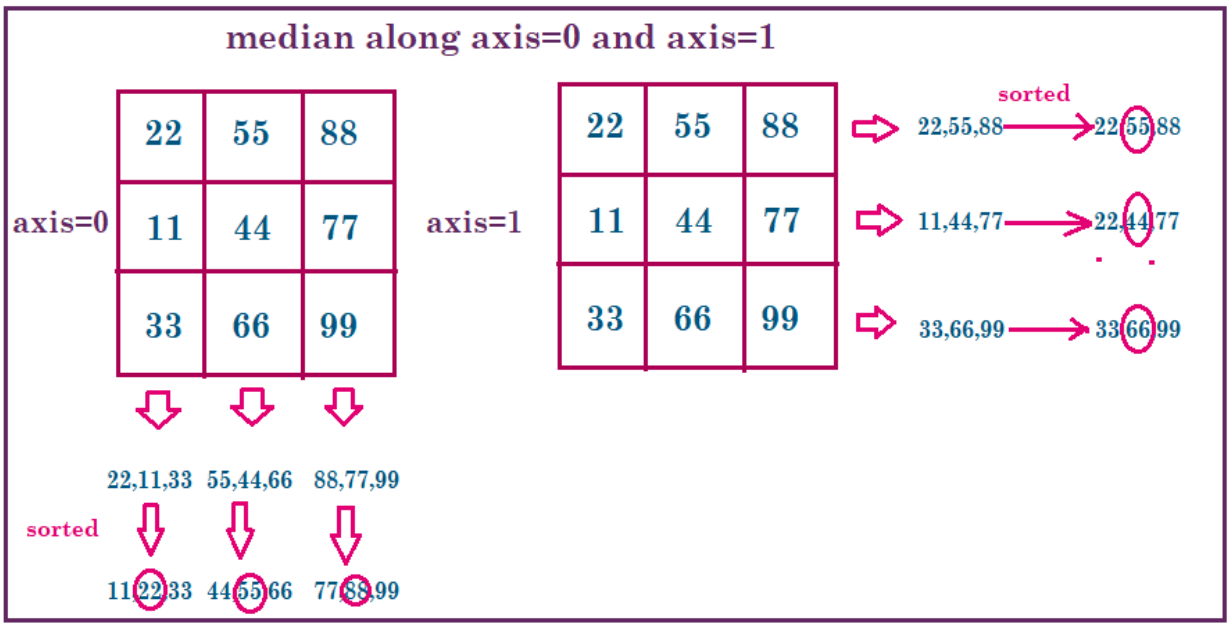
- **axis=None(default)** - The array is flattened to 1-D array(sorted) and find the median value
- **axis=0** - rows. Consider columns with all rows and find the median
- **axis=1** - columns. Consider rows with all columns and find the median



In [480]:

```
# 2-D array median
a = np.arange(9).reshape(3,3)
print(f"The original 2-D array(already sorted) : \n {a}")
print(f"Mean of the 2-D array along axis=None : {np.median(a)}")
print(f"Mean of the 2-D array along axis=0 : {np.median(a,axis=0)}")
print(f"Mean of the 2-D array along axis=1 : {np.median(a,axis=1)}")
```

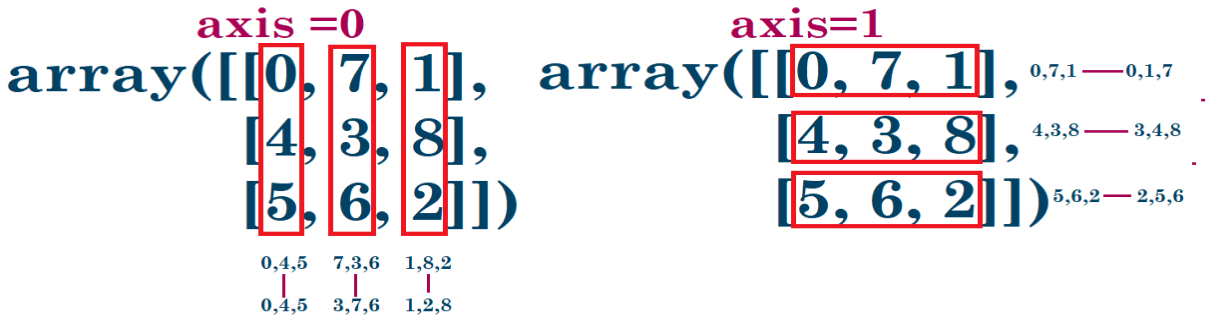
```
The original 2-D array(already sorted) :
 [[0 1 2]
 [3 4 5]
 [6 7 8]]
Mean of the 2-D array along axis=None : 4.0
Mean of the 2-D array along axis=0 : [3. 4. 5.]
Mean of the 2-D array along axis=1 : [1. 4. 7.]
```



In [481]:

```
# 2-D array median ==> unsorted elements
a = np.array([[22,55,88],[11,44,55],[33,66,99]])
print(f"The original 2-D array(unsorted) : \n {a}")
print(f"Mean of the 2-D array along axis=None : {np.median(a)}")
print(f"Mean of the 2-D array along axis=0 : {np.median(a,axis=0)}")
print(f"Mean of the 2-D array along axis=1 : {np.median(a,axis=1)}")
```

```
The original 2-D array(unsorted) :
 [[22 55 88]
 [11 44 55]
 [33 66 99]]
Mean of the 2-D array along axis=None : 55.0
Mean of the 2-D array along axis=0 : [22. 55. 88.]
Mean of the 2-D array along axis=1 : [55. 44. 66.]
```



In [482]:

```
# 2-D array median ==> unsorted elements using shuffle
a = np.arange(9)
np.random.shuffle(a)
a = a.reshape(3,3)
print(f"The original 2-D array(unsorted) : \n {a}")
print(f"Mean of the 2-D array along axis=None : {np.median(a)}")
print(f"Mean of the 2-D array along axis=0 : {np.median(a,axis=0)}")
print(f"Mean of the 2-D array along axis=1 : {np.median(a,axis=1)}")
```

```
The original 2-D array(unsorted) :
 [[6 8 4]
 [3 0 5]
 [2 1 7]]
Mean of the 2-D array along axis=None : 4.0
Mean of the 2-D array along axis=0 : [3. 1. 5.]
Mean of the 2-D array along axis=1 : [6. 3. 2.]
```

Variance value

np.var(a)

a.var()

The variance is a measure of variability. It is calculated by taking the average of squared deviations from the mean.

- **average of**
- **squared**
- **deviations from the mean.**

In [483]:

```
import numpy as np
help(np.var)
```

```
Help on function var in module numpy:

var(a, axis=None, dtype=None, out=None, ddof=0, keepdims=<no value>, *, where
=<no value>)
    Compute the variance along the specified axis.

    Returns the variance of the array elements, a measure of the spread of a
    distribution.  The variance is computed for the flattened array by
    default, otherwise over the specified axis.
```

The variance is a measure of variability. It is calculated by taking the average of squared deviations from the mean.

a = [1,2,3,4,5]

Original Array (x)	Deviation (x-Mean)	Square of Deviation
1	1-3 = -2	4
2	2-3 = -1	1
3	3-3= 0	0
4	4-3 = 1	1
5	5-3 = 2	4

Mean = $\sum x / n$

mean(a) = (1+2+3+4+5)/5 = 3.0

deviations from the mean:
[-2.0,-1.0,0.0,1.0,2.0]

squares of deviations from the mean:
[4.0,1.0,0.0,1.0,4.0]

Average of squares of deviations from the mean: (4+1+0+1+4)/5 =2.0 VARIANCE

1-D array

In [484]:

```
a = np.array([1,2,3,4,5])
print(f"Original 1-D array : {a}")
print(f"Variance of 1-D array unsing np.var(a): {np.var(a)}")
print(f"Variance of 1-D array unsing a.var(): {a.var()}")
```

```
Original 1-D array : [1 2 3 4 5]
Variance of 1-D array unsing np.var(a): 2.0
Variance of 1-D array unsing a.var(): 2.0
```

2-D array

- **axis=None(default)** - The array is flattened to 1-D array(sorted) and find the variance value
- **axis=0** - rows. Consider columns with all rows and find the variance
- **axis=1** - columns. Consider rows with all columns and find the variance

In [485]:

```
a = np.arange(6).reshape(2,3)
print(f"Original 2-D array :\n {a}")
print(f"Variance of 2-D array using np.var(a) along axis=None: {np.var(a)}")
print(f"Variance of 2-D array using np.var(a) along axis=0: {np.var(a,axis=0)}")
print(f"Variance of 2-D array using np.var(a) along axis=1: {np.var(a,axis=1)}")
```

```
Original 2-D array :
 [[0 1 2]
 [3 4 5]]
Variance of 2-D array using np.var(a) along axis=None: 2.9166666666666665
Variance of 2-D array using np.var(a) along axis=0: [2.25 2.25 2.25]
Variance of 2-D array using np.var(a) along axis=1: [0.66666667 0.66666667]
```

Standard Deviation value

- **np.std(a)**
- **a.std()**
- Variance means the average of squares of deviations from the mean.
- Standard deviation is the square root of the variance.

1-D array

In [486]:

```
import math
a = np.array([1,2,3,4,5])
print(f"Original 1-D array : {a}")
print(f"Variance of 1-D array unsing np.var(a): {np.var(a)}")
print(f"Standard Deviation  of 1-D array unsing np.std(a): {np.std(a)}")
print(f"Square root of Variannce : {math.sqrt(np.var(a))}")
```

```
Original 1-D array : [1 2 3 4 5]
Variance of 1-D array unsing np.var(a): 2.0
Standard Deviation  of 1-D array unsing np.std(a): 1.4142135623730951
Square root of Variannce : 1.4142135623730951
```

2-D array

In [487]:

```
import math
a = np.arange(6).reshape(2,3)
print(f"Original 2-D array :\n {a}")
print("*"*100)
print(f"Variance of 2-D array using np.var(a) along axis=None: {np.var(a)}")
print(f"Standard Deviation of 2-D array using np.std(a) along axis=None:
{np.std(a)}")
print(f"Square root of Variannce : {math.sqrt(np.var(a))}")
print("*"*100)
print(f"Variance of 2-D array using np.var(a) along axis=0: {np.var(a,axis=0)}")
print(f"Standard Deviation of 2-D array using np.std(a) along axis=0:
{np.std(a,axis=0)}")
print("*"*100)
print(f"Variance of 2-D array using np.var(a) along axis=1: {np.var(a,axis=1)}")
print(f"Standard Deviation of 2-D array using np.std(a) along axis=1:
{np.std(a,axis=1)}")
print("*"*100)
```

```
Original 2-D array :
 [[0 1 2]
 [3 4 5]]
****************************************************************************
Variance of 2-D array using np.var(a) along axis=None: 2.9166666666666665
Standard Deviation of 2-D array using np.std(a) along axis=None: 1.7078251276
59933
Square root of Variannce : 1.707825127659933
****************************************************************************
Variance of 2-D array using np.var(a) along axis=0: [2.25 2.25 2.25]
Standard Deviation of 2-D array using np.std(a) along axis=0: [1.5 1.5 1.5]
****************************************************************************
Variance of 2-D array using np.var(a) along axis=1: [0.66666667 0.66666667]
Standard Deviation of 2-D array using np.std(a) along axis=1: [0.81649658 0.8
1649658]
****************************************************************************
```

Summary

- **np.min(a)/np.amin(a)/a.min()**--->Returns the minimum value of the array
- **np.max(a)/np.amax(a)/a.max()**--->Returns the maximum value of the array
- **np.sum(a)/a.sum()**--->Returns the Sum of values of the array
- **np.mean(a)/a.mean()**--->Returns the arithmetic mean of the array.
- **np.median(a)** --->Returns median value of the array
- **np.var(a)/a.var()** --->Returns variance of the values in the array
- **np.std(a)/a.std()** --->Returns Standard deviation of the values in the array

Chapter-21
Numpy Mathematical Functions

Numpy Mathematical Functions

- The **functions** which operates **element by element on whole array**, are called **universal functions.**
- To perform mathematical operations numpy library contains several universal functions **(ufunc).**
- **np.exp(a)** --->Takes e to the power of each value. e value: 2.7182
- **np.sqrt(a)** --->Returns square root of each value.
- **np.log(a)** --->Returns logarithm of each value.
- **np.sin(a)** ---->Returns the sine of each value.
- **np.cos(a)** --->Returns the co-sine of each value.
- **np.tan(a)** --->Returns the tangent of each value.

In [488]:

```
import numpy as np
import math
a = np.array([[1,2],[3,4]])
print(f"np.exp(a) value :\n {np.exp(a)}")
print(f"Value of e power 1 ==> {math.exp(1)} ")
print(f"Value of e power 2 ==> {math.exp(2)} ")
print(f"Value of e power 3 ==> {math.exp(3)} ")
print(f"Value of e power 4 ==> {math.exp(4)} ")
```

```
np.exp(a) value :
 [[ 2.71828183  7.3890561 ]
 [20.08553692 54.59815003]]
Value of e power 1 ==> 2.718281828459045
Value of e power 2 ==> 7.38905609893065
Value of e power 3 ==> 20.085536923187668
Value of e power 4 ==> 54.598150033144236
```

In [489]:

```
# Mathematical functions
a = np.arange(5)
print(f"exp() on a  ==> {np.exp(a)}")
```

```
print(f"sqrt() on a ==> {np.sqrt(a)}")
print(f"sin() on a  ==> { np.sin(a)}")
print(f"cos() on a  ==> {np.cos(a)}")
print(f"tan() on a  ==> {np.tan(a)}")
print(f"log() on a :{np.log(a)}")
```

```
exp() on a  ==> [ 1.          2.71828183  7.3890561  20.08553692 54.59815003]
sqrt() on a ==> [0.         1.         1.41421356 1.73205081 2.        ]
sin() on a  ==> [ 0.          0.84147098  0.90929743  0.14112001 -0.7568025 ]
cos() on a  ==> [ 1.          0.54030231 -0.41614684 -0.9899925  -0.65364362]
tan() on a  ==> [ 0.          1.55740772 -2.18503986 -0.14254654  1.15782128]
log() on a :[      -inf 0.         0.69314718 1.09861229 1.38629436]

<ipython-input-489-9de2ffb7f031>:8: RuntimeWarning: divide by zero encountere
d in log
  print(f"log() on a :{np.log(a)}")
```

Chapter-22
How to find unique items and count

How to find unique items and count

using unique() function we can find the unique items and their count in ndarray

unique(ar, return_index=False, return_inverse=False, return_counts=False, axis=None) ==> Find the unique elements of an array.

Returns the sorted unique elements of an array.

There are three optional outputs in addition to the unique elements:

- the indices of the input array that give the unique values ==> **return_index**
- the indices of the unique array that reconstruct the input array
- the number of times each unique value comes up in the input array ==> **return_counts**

In [490]:

```
import numpy as np
help(np.unique)
```

```
Help on function unique in module numpy:

unique(ar, return_index=False, return_inverse=False, return_counts=False, axi
s=None)
    Find the unique elements of an array.
```

Get array with unique elements

In [491]:

```
a = np.array([1,1,2,3,4,2,3,4,4,1,2,3,4,5,5,6])
print(f"Original array : {a}")
print(f"Unique elements in the  array using np.unique(a) : {np.unique(a)}")
```

```
Original array : [1 1 2 3 4 2 3 4 4 1 2 3 4 5 5 6]
Unique elements in the  array using np.unique(a) : [1 2 3 4 5 6]
```

Get indices also

```
In [492]:

a = np.array([1,1,2,3,4,2,3,4,4,1,2,3,4,5,5,6])
items,indices = np.unique(a,return_index=True)
print(f"Original array : {a}")
print(f"Unique Elements :{items}")
print(f"indices of unique elements: {indices}")
```

```
Original array : [1 1 2 3 4 2 3 4 4 1 2 3 4 5 5 6]
Unique Elements :[1 2 3 4 5 6]
indices of unique elements: [ 0  2  3  4 13 15]
```

To get count also:

```
In [493]:

a = np.array([1,1,2,3,4,2,3,4,4,1,2,3,4,5,5,6])
items,counts = np.unique(a,return_counts=True)
print(f"Original array : {a}")
print(f"Unique Elements :{items}")
print(f"count of unique elements: {counts}")
```

```
Original array : [1 1 2 3 4 2 3 4 4 1 2 3 4 5 5 6]
Unique Elements :[1 2 3 4 5 6]
count of unique elements: [3 3 3 4 2 1]
```

To get all:

```
In [494]:

a = np.array([1,1,2,3,4,2,3,4,4,1,2,3,4,5,5,6])
items,indices,counts = np.unique(a,return_index=True,return_counts=True)
print(f"Original array : {a}")
print(f"Unique Elements :{items}")
print(f"indices of unique elements: {indices}")
print(f"count of unique elements: {counts}")
```

```
Original array : [1 1 2 3 4 2 3 4 4 1 2 3 4 5 5 6]
Unique Elements :[1 2 3 4 5 6]
indices of unique elements: [ 0  2  3  4 13 15]
count of unique elements: [3 3 3 4 2 1]
```

In [495]:

```
# To get all in a program

import numpy as np
a = np.array(['a','a','b','c','a','a','b','c','a','b','d'])
items,indices,counts = np.unique(a,return_index=True,return_counts=True)
for item,index,count in zip(np.nditer(items),np.nditer(indices),np.nditer(counts)):
   print(f"Element '{item}' occurred {count} times and its first occurrence
index:{index}")
```

```
Element 'a' occurred 5 times and its first occurrence index:0
Element 'b' occurred 3 times and its first occurrence index:2
Element 'c' occurred 2 times and its first occurrence index:3
Element 'd' occurred 1 times and its first occurrence index:10
```

Chapter-23
Numpy Practice Quesions Set-1

Numpy Practice Quesions Set-1

Q1. Create an array of 7 zeros?

```
In [496]:

import numpy as np
# np.zeros(7)
# np.zeros(7,dtype=int)
# np.full(7,0)
print(f"Array of 7 zeros using np.zeros(7) : {np.zeros(7)}")
print(f"Array of 7 zeros using np.zeros(7,dtype=int) : {np.zeros(7,dtype=int)}")
print(f"Array of 7 zeros using np.full(7,0) : {np.full(7,0)}")
```

```
Array of 7 zeros using np.zeros(7) : [0. 0. 0. 0. 0. 0. 0.]
Array of 7 zeros using np.zeros(7,dtype=int) : [0 0 0 0 0 0 0]
Array of 7 zeros using np.full(7,0) : [0 0 0 0 0 0 0]
```

Q2. Create an array of 9 ones?

```
In [497]:

# np.ones(9)
# np.ones(9,dtype=int)
# np.full(9,1)
print(f"Array of 9 ones using np.ones(9) : {np.ones(9)}")
print(f"Array of 9 ones using np.ones(9,dtype=int) : {np.ones(9,dtype=int)}")
print(f"Array of 9 ones using np.full(9,1) : {np.full(9,1)}")
```

```
Array of 9 ones using np.ones(9) : [1. 1. 1. 1. 1. 1. 1. 1. 1.]
Array of 9 ones using np.ones(9,dtype=int) : [1 1 1 1 1 1 1 1 1]
Array of 9 ones using np.full(9,1) : [1 1 1 1 1 1 1 1 1]
```

Q3. Create an array of 10 fours?

In [498]:

```
# np.full(10,4)
# np.zeros(10,dtype=int)+4
# np.ones(10,dtype=int)+3
print(np.full(10,4))
print(np.zeros(10,dtype=int)+4)
print(np.ones(10,dtype=int)+3)
```

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

Q4. Create an array of integers from 10 to 40?

In [499]:

```
print(np.arange(10,41))
```

```
[10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33
 34 35 36 37 38 39 40]
```

Q5. Create an array of integers from 10 to 40 which are even?

In [500]:

```
np.arange(10,41,2)
```

Out[500]:

```
array([10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40])
```

Q6. Create an array of integers from 10 to 40 which are odd?

In [501]:

```
np.arange(11,41,2)
```

Out[501]:

```
array([11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31, 33, 35, 37, 39])
```

Q7. Create an array of integers from 10 to 40 which are divisible by 7?

In [502]:

```
# 1st way:
print(np.arange(14,41,7))
```

```
[14 21 28 35]
```

In [503]:

```
# 2nd way:
a = np.arange(10,41)
print(a)
print(a[a%7==0])
```

```
[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]
[14 21 28 35]
```

Q8. Create a numpy array having 10 numbers starts from 24 but only even numbers?

In [504]:

```
# 1st way:
print(f"using np.arange(24,43,2): {np.arange(24,43,2)}")
# 2nd way:
a = np.arange(24,50)
a = a[a%2==0]
print(f"using np.resize(a,10) : {np.resize(a,10)}")
```

```
using np.arange(24,43,2): [24 26 28 30 32 34 36 38 40 42]
using np.resize(a,10) : [24 26 28 30 32 34 36 38 40 42]
```

Q9. Create a 4X4 matrix with elements from 1 to 16?

In [505]:

```
np.arange(1,17).reshape(4,4)
```

Out[505]:

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

Q10. Create a 4X4 identity matrix?

In [506]:

```
print(f"using np.eye(4) : \n {np.eye(4)}")
print(f"using np.identity(4) :\n {np.identity(4)}")
```

```
using np.eye(4) :
 [[1. 0. 0. 0.]
 [0. 1. 0. 0.]
 [0. 0. 1. 0.]
 [0. 0. 0. 1.]]
using np.identity(4) :
 [[1. 0. 0. 0.]
 [0. 1. 0. 0.]
 [0. 0. 1. 0.]
 [0. 0. 0. 1.]]
```

Q11. By using numpy module, generate ndarray with random numbers from 1 to 100 with the shape:(2,3,4)?

In [507]:

```
np.random.randint(1,101,size=(2,3,4))
```

Out[507]:

```
array([[[51, 52, 93, 82],
        [ 3, 83, 48, 19],
        [96, 39, 44, 98]],

       [[53, 52, 11, 20],
        [76, 14, 89, 50],
        [69, 65, 71, 65]]])
```

Q12. By using numpy module, generate a random number between 0 and 1?

In [508]:

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

Out[508]:

```
0.8054592849192298
```

Q13. By using numpy module, generate an array of 10 random samples from a uniform distribution over [0,1)?

In [509]:

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

Out[509]:

```
array([0.57786998, 0.21131385, 0.47052737, 0.11353636, 0.07344479,
       0.12596435, 0.51560813, 0.00377141, 0.51092043, 0.51462955])
```

Q14. By using numpy module, generate an array of 10 random samples from a uniform distribution over [10,20)?

In [510]:

```
# np.random.uniform(low=0.0,high=1.0,size=None)
np.random.uniform(10,20,10)
```

Out[510]:

```
array([12.28628812, 17.37509048, 18.88105358, 13.83690349, 15.83230432,
       12.12287601, 12.81889152, 17.77903863, 12.94707042, 13.30409307])
```

Q15. By using numpy module, generate an array of 10 random samples from a standard normal distribution of mean 0 and standard deviation 1?

In [511]:

```
np.random.randn(10)
```

Out[511]:

```
array([ 0.71579249,  1.42716022,  0.31241851, -0.37037978,  1.82957797,
       -2.40593885,  0.62470205, -1.58715091,  0.16443136,  0.54251342])
```

Q16. By using numpy module, generate an array of 10 random samples from a standard normal distribution of mean 15 and and stadard deviation 4?

In [512]:

```
# normal(loc=0.0, scale=1.0, size=None)
# loc--->mean
# scale--->standard deviation
```

a = np.random.normal(15,4,10)
a

Out[512]:

```
array([18.29936929, 11.48075769, 11.01075089, 23.83089517, 18.43564922,
       16.47381834, 14.684902  , 15.13868843, 11.67061641, 16.88892347])
```

Q17. Create an array of 10 linearly spaced points between 1 and 100?

In [513]:

np.linspace(1,100,10,retstep=True)

Out[513]:

```
(array([  1.,  12.,  23.,  34.,  45.,  56.,  67.,  78.,  89., 100.]), 11.0)
```

Q18. Create an array of 15 linearly spaced points between 0 and 1?

In [514]:

np.linspace(0,1,15,retstep=True)

Out[514]:

```
(array([0.        , 0.07142857, 0.14285714, 0.21428571, 0.28571429,
        0.35714286, 0.42857143, 0.5       , 0.57142857, 0.64285714,
        0.71428571, 0.78571429, 0.85714286, 0.92857143, 1.        ]),
 0.07142857142857142)
```

Chapter-24
Numpy Practice Quesions Set-2

Numpy Practice Quesions Set-2

In [515]:

```
import numpy as np
a = np.arange(1,37).reshape(6,6)
a
```

Out[515]:

```
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]])
```

Q1. How to access element 16?

In [516]:

```
print(f"a[2][3] ==> {a[2][3]}")
print(f"a[2,3] ==> {a[2,3]}")
```

```
a[2][3] ==> 16
a[2,3] ==> 16
```

Q2. To get the following array: array([[2, 3, 4, 5]])

In [517]:

```
oned_array = a[0,1:5]
twod_array = a[0:1,1:5]
print(f"using with indexing : {oned_array}")
print(f"using with slicing : {twod_array}")
```

```
using with indexing : [2 3 4 5]
using with slicing : [[2 3 4 5]]
```

Q3. To get the following array: array([[2, 3, 4, 5]])

In [518]:

```
oned_array = a[0,1:5]
twod_array = a[0:1,1:5]
print(f"using with indexing : {oned_array}")
print(f"using with slicing : {twod_array}")
```

```
using with indexing : [2 3 4 5]
using with slicing : [[2 3 4 5]]
```

Q4. To get the following array

```
array([[ 1,  2,  3,  4,  5,  6],

       [31, 32, 33, 34, 35, 36]])
```

In [519]:

```
a[::5,:]
```

Out[519]:

```
array([[ 1,  2,  3,  4,  5,  6],
       [31, 32, 33, 34, 35, 36]])
```

Q5. To get the following array

```
array([[ 9, 10],

      [15, 16],

      [21, 22]])
```

In [520]:

```
a[1:4,2:4]
```

Out[520]:

```
array([[ 9, 10],
       [15, 16],
       [21, 22]])
```

Q6. Create 1-D array with all even numbers of a?

In [521]:

```
a[a%2==0]
```

Out[521]:

```
array([ 2,  4,  6,  8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34,
       36])
```

Q7. Create a 1-D array with elements of a which are divisible by 5?

In [522]:

```
a[a%5==0]
```

Out[522]:

```
array([ 5, 10, 15, 20, 25, 30, 35])
```

Q8. Create 1-D array with elements 8,17,26 and 35. We have to use elements of a?

In [523]:

```
# a[[row_indices],[column_indices]]
a[[1,2,4,5],[1,4,1,4]]
```

Out[523]:

```
array([ 8, 17, 26, 35])
```

Q9. Select minimum element of this ndarray?

In [524]:

```
# np.amin(a)
# np.min(a)
# a.min()
print(f"np.amin(a) ==> {np.amin(a)}")
print(f"np.min(a) ==> {np.min(a)}")
print(f"a.min() ==> {a.min()}")
```

```
np.amin(a) ==> 1
np.min(a) ==> 1
a.min() ==> 1
```

Q10. Select maximum element of this ndarray?

In [525]:

```
# np.amax(a)
# np.max(a)
# a.max()
print(f"np.amax(a) ==> {np.amax(a)}")
print(f"np.max(a) ==> {np.max(a)}")
print(f"a.max() ==> {a.max()}")
```

```
np.amax(a) ==> 36
np.max(a) ==> 36
a.max() ==> 36
```

Q11. Find sum of elements present inside this array?

In [526]:

```
print(f"np.sum(a) ==> {np.sum(a)}")
print(f"a.sum() ==> {a.sum()}")
```

```
np.sum(a) ==> 666
a.sum() ==> 666
```

Q12. Find sum of elements along axis-0?

In [527]:

```
print(f"np.sum(a,axis=0) ==> {np.sum(a,axis=0)}")
print(f"a.sum(axis=0) ==> {a.sum(axis=0)}")
```

```
np.sum(a,axis=0) ==> [ 96 102 108 114 120 126]
a.sum(axis=0) ==> [ 96 102 108 114 120 126]
```

Q13. Find mean of this array?

In [528]:

```
print(f"np.mean(a) ==> {np.mean(a)}")
print(f"a.mean() ==> {a.mean()}")
```

```
np.mean(a) ==> 18.5
a.mean() ==> 18.5
```

Q14. Find median of this array?

In [529]:

```
print(f"np.median(a) ==> {np.median(a)}")
```

```
np.median(a) ==> 18.5
```

Q15. Find variance of this array?

In [530]:

```
print(f"np.var(a) ==> {np.var(a)}")
print(f"a.var() ==> {a.var()}")
```

```
np.var(a) ==> 107.91666666666667
a.var() ==> 107.91666666666667
```

Q16. Find standard deviation of this array?

In [531]:

```
print(f"np.std(a) ==> {np.std(a)}")
print(f"a.std() ==> {a.std()}")
```

```
np.std(a) ==> 10.388294694831615
a.std() ==> 10.388294694831615
```

Chapter-25
Numpy Quiz Questions

Numpy Quiz Questions

Q1. Consider the following code: `import numpy as np a = np.array([]) print(a.shape)`

What is the output ? </code>

- **A. (0,)**
- B. (1,)
- C. (1,1)
- D. 0

Ans : A

In [532]:

```
import numpy as np
a = np.array([])
print(a.shape)
```

```
(0,)
```

Q2. Consider the following code: `import numpy as np a = np.arange(10,20,-1) b = a.reshape(5,2) print(b.flatten())`

- A. [10 11 12 13 14 15 16 17 18 19]
- B. [20 19 18 17 16 15 14 13 12 11]
- C. array([])
- **D. ValueError**

Ans: D

In [533]:

```
import numpy as np
a = np.arange(10,20,-1)
b = a.reshape(5,2)
print(b.flatten())
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-533-d495ed6809f0> in <module>
      1 import numpy as np
      2 a = np.arange(10,20,-1)
----> 3 b = a.reshape(5,2)
      4 print(b.flatten())

ValueError: cannot reshape array of size 0 into shape (5,2)
```

Q3. Consider the following code:

```
import numpy as np
a = np.arange(1,6)
a = a[::-2]
print(a)
```

What is the output ?

- A. [1 2 3 4 5]
- B. [6 4 2]
- **C. [5 3 1]**
- D. [4 2]

Ans: C

In [534]:

```
import numpy as np
a = np.arange(1,6)
a = a[::-2]
print(a)
```

```
[5 3 1]
```

Q4. Consider the following code:

```
a = np.array([3,3])
b = np.array([3,3.5])
c = np.array([3,'3'])
d = np.array([3,True])
```

The dtypes of a,b,c and d are:

- **A. int,float,str,int**
- B. int,int,str,bool
- C. float,float,int,int
- D. int,float,str,bool
- E. ValueError while creating b,c,d

Ans: A

In [535]:

```
a = np.array([3,3])
b = np.array([3,3.5])
c = np.array([3,'3'])
d = np.array([3,True])
print(f"a dtype =>{a.dtype}")
print(f"b dtype =>{b.dtype}")
print(f"c dtype =>{c.dtype}")
print(f"d dtype =>{d.dtype}")
```

```
a dtype =>int32
b dtype =>float64
c dtype =><U11
d dtype =>int32
```

Q5. Consider the code:

```
a = np.array([1,2,3,2,3,4,5,2,3,4,1,2,3,6,7])
print(a[2:5])
```

What is the output?

- **A. [3 2 3]**
- B. [2 3 2]
- C. [3 2 3 4]
- D. IndexErrror

Ans: A

In [536]:

```
a = np.array([1,2,3,2,3,4,5,2,3,4,1,2,3,6,7])
print(a[2:5])
```

```
[3 2 3]
```

Q6. Consider the code:

```
a = np.array([1,2,3,2,3,4,5,2,3,4,1,2,3,6,7])
print(a[2:7:2])
```

What is the output?

- **A. [3 3 5]**
- B. [3 2 3 4 5]
- C. [3 2 3 4]
- D. IndexErrror

Ans: A

In [537]:

```
a = np.array([1,2,3,2,3,4,5,2,3,4,1,2,3,6,7])
print(a[2:7:2])
```

```
[3 3 5]
```

Q7. Consider the code:

```
a = np.array([1,2,3,2,3,4,5,2,3,4,1,2,3,6,7])
print(a[:3:3])
```

What is the output?

- **A. [1]**
- B. [1 2 3]
- C. [1 3]
- D. IndexErrror

Ans: A

In [538]:

```
a = np.array([1,2,3,2,3,4,5,2,3,4,1,2,3,6,7])
print(a[:3:3])
```

```
[1]
```

Q8. Consider the code:

```
a = np.array([1,2,3,2,3,4,5,2,3,4,1,2,3,6,7])
print(a[7:2:2])
```

What is the output?

- A. [2 4 2]
- B. [2 5 4 3]
- **C. []**
- D. IndexErrror

Ans: C

In [539]:

```
a = np.array([1,2,3,2,3,4,5,2,3,4,1,2,3,6,7])
print(a[7:2:2])
```

```
[]
```

Q9. Consider the code:

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

What is the output?

- A. [1 2 4]
- **B. [2 3 3]**
- C. [True True True]
- D. IndexErrror

Ans: B

In [540]:

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

```
[2 3 3]
```

Q10. Consider the ndarray:

```
a = np.arange(20).reshape(5,4)
```

Which of the following options will provide 15?

- A. a[3][3]
- B. a[-2][-1]
- C. a[3][-1]
- D. a[3,3]
- **E. All of these**

Ans: E

In [541]:

```
a = np.arange(20).reshape(5,4)
print(f" value of a[3][3] ==> {a[3][3]}")
print(f" value of a[-2][-1] ==> {a[-2][-1]}")
print(f" value of a[3][-1] ==> {a[3][-1]}")
print(f" value of a[3,3] ==> {a[3,3]}")
```

```
 value of a[3][3] ==> 15
 value of a[-2][-1] ==> 15
 value of a[3][-1] ==> 15
 value of a[3,3] ==> 15
```

Q11. We can create Numpy array only to represent 2 Dimensional matrix?

- A. True
- **B. False**

Ans: B

Q12. Which of the following is valid way of importing numpy library?

- A. from numpy import np
- **B. import numpy as np**
- **C. import numpy as np1**
- D. import np as numpy

Ans: B and C

Q13. Which of the following cannot be used as numpy index?

- A. 0
- B. 1
- C. -1
- D. -2
- **E. None of these**

Ans: E

Q14. Which of the following is correct syntax to create a numpy array?

- **A. np.array([10,20,30,40])**
- B. np.createArray([10,20,30,40])
- C. np.makeArray([10,20,30,40])
- D. np([10,20,30,40])

Ans: A

Q15. Which of the following are valid ways of finding the number of dimensions of input array a?

- **A. np.ndim(a)**
- B. np.dim(a)
- C. a.ndim()
- **D. a.ndim**

Ans: A and D

Q16. Consider the array

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

Which of the following prints first element of this array?

- **A. print(a[0])**
- B. print(a[1])
- C. print(a.0)
- D. print(a.1)

Ans: A

Q17. How to check data type of ndarray a?

- A. np.dtype(a)
- B. a.dtype()
- **C. a.dtype**
- D. All of these

Ans: C

Q18. Which of the following is valid of creating float type ndarray?

- **A. a = np.array([10,20,30,40],dtype='f')**
- **B. a = np.array([10,20,30,40],dtype='float')**
- **C. a = np.array([10,20,30,40],dtype=float)**
- D. a = np.array([10,20,30,40])

Ans: A,B,C

Q19. Which of the following statements are valid?

- **A. If we perform any changes to the original array, then those changes will be reflected to the VIEW.**
- B. If we perform any changes to the original array, then those changes won't be reflected to the VIEW.
- C. If we perform any changes to the original array, then those changes will be reflected to the COPY.
- **D. If we perform any changes to the original array, then those changes won't be reflected to the COPY.**

Ans: A,D

Q20. Choose only one appropriate statement regarding shape of ndarray?

- **A. The shape represents the size of each dimension.**
- B. The shape represents only the number of rows.
- C. The shape represents only the number of columns.
- D. The shape represents the total size of the array.

Ans: A

Q21. By using which functions of numpy library, we can perform search operation for the required elements?

- A. find()
- B. search()
- **C. where()**
- D. All of these

Ans: C

Q22. Consider the following array: `a = np.array([10,20,30,10,20,10,10,30,40])`
`Which of the following code represents the indices where element 10 present?`

- A. np.find(a == 10)
- **B. np.where(a == 10)**
- C. np.search(a == 10)
- D. None of these

Ans: B

Q23. Which of the following code collects samples from uniform distribution of 1000 values in the interval `[10,100)`?

- A. np.uniform(10,100,size=1000)
- **B. np.random.uniform(10,100,size=1000)**
- **C. np.random.uniform(low=10,high=100,size=1000)**
- D. np.random.uniform(from=10,to=100,size=1000)

Ans: B,C

- np.random.rand()--->uniform distribution over [0,1)
- np.random.uniform()--->uniform distribution over [a,b)
- uniform(low=0.0, high=1.0, size=None)

Q24. Which of the following code collects samples from normal distribution of 1000 values with the mean 10 and standard deviation 0.3?

- A. np.normal(10,0.3,1000)
- **B. np.random.normal(10,0.3,1000)**
- C. np.random.normal(mean=10,std=0.3,size=1000)
- **D. np.random.normal(loc=10,scale=0.3,size=1000)**

Ans: B and D

- **normal(loc=0.0, scale=1.0, size=None)**
- **loc** : float or array_like of floats **Mean**("centre") of the distribution.
- **scale** : float or array_like of floats
  Standard deviation (spread or "width") of the distribution. Must be non-negative.

Q25. Which of the following is valid of performing add operation at element level of two ndarrays a and b?

- **A. np.add(a,b)**
- B. np.sum(a,b)
- C. np.append(a,b)
- **D. a+b**

Ans: A,D

Q26. Which of the following is valid of performing subtract operation at element level of two ndarrays a and b?

- **A. a-b**
- B. np.minus(a,b)
- C. np.min(a,b)
- **D. np.subtract(a,b)**

Ans: A,D

Q27. Which of the following returns 1.0 if a = 1.2345?

- A. np.trunc(a)
- B. np.fix(a)
- C. np.around(a)
- **D. All of these**

Ans: D

np.trunc(a)

- Remove the digits after decimal point

In [542]:

```
print(np.trunc(1.23456))
print(np.trunc(1.99999))
```

```
1.0
1.0
```

np.fix(a)

- Round to nearest integer towards zero

In [543]:

```
print(np.fix(1.234546))
print(np.fix(1.99999999))
```

```
1.0
1.0
```

np.around(a)

- It will perform round operation.
- If the next digit is >=5 then remove that digit by incrementing previous digit.
- If the next digit is <5 then remove that digit and we are not required to do anything with the previous digit.

In [544]:

```
print(np.around(1.23456))
print(np.around(1.99999))
```

```
1.0
2.0
```

Q28. Which of the following returns 2.0 if a = 1.995?

- A. np.trunc(a)
- B. np.fix(a)
- **C. np.around(a)**
- D. All of these

Ans: C

Q29. Which of the following are valid ways of creating a 2-D array?

- **A. np.array([[10,20,30],[40,50,60]])**
- B. np.array([10,20,30,40,50,60])
- C. np.array([10,20,30,40,50,60],ndim=2)
- **D. np.array([10,20,30,40,50,60],ndmin=2)**

Ans: A,D

Q30. Consider the code: `a = np.array([10,20,30,40]) print(np.cumsum(a))`

What is the result? </code>

- A. [10 20 30 40]
- **B. [10 30 60 100]**
- C. [100 100 100 100]
- D. None of these

Ans: B

In [545]:

help(np.cumsum)

```
Help on function cumsum in module numpy:

cumsum(a, axis=None, dtype=None, out=None)
    Return the cumulative sum of the elements along a given axis.
```

In [546]:

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

```
[ 10  30  60 100]
```

Q31. Consider the array: `a = np.array([10, 15, 20, 25, 30, 35, 40])`

Which of the following selects items from the second item to fourth item?

- **A. a[1:4]**
- B. a[1:5]
- C. a[2:5]
- D. a[2:4]

Ans: A

Q32. Consider the array:

```
a = np.array([10, 15, 20, 25, 30, 35, 40])
```

Which of the following selects items from the third item to fourth item?

- A. a[1:4]
- B. a[1:5]
- C. a[2:5]
- **D. a[2:4]**

Ans: D

Q33. Consider the array:

```
a = np.array([10, 15, 20, 25, 30, 35, 40])
```

Which of the following selects every other item from the second item to sixth item?

- **A. a[1:6:2]**
- B. a[1:5:2]
- C. a[2:6:2]
- D. a[2:7:2]

Ans: A

Q34. Consider the array:

```
a = np.array([10, 15, 20, 25, 30, 35, 40])
```

Which of the following selects every other item from total array?

- A. a[:7:2]
- B. a[0::2]
- C. a[0:7:2]
- D. a[::2]
- **E. All of these**

Ans: E

Q35. Consider the following array:

```
a = np.array([10.5,20.6,30.7])
```

Which of the following is the valid way to convert array into int data type? </code>

- A. newarray = a.int()
- B. newarray = a.asInt()
- **C. newarray = a.astype(int)**
- **D. newarray = a.astype('int')**
- **E. newarray = np.int32(a)**

Ans: C,D,E

Q36. Consider the following array:

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

```
Which of the following is valid way to get element 50?
```

- A. a[1][1]
- B. a[-1][-2]
- C. a[1][-2]
- D. a[-1][1]
- **E. All the above**

Ans: E

Q37. Consider the following array:

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

```
Which of the following is valid way to get element 30?
```

- A. a[0][2]
- B. a[-2][-1]
- C. a[0][-1]
- D. a[-2][2]
- **E. All the above**

Ans: E