Data analysis With NumPy

Data analysisi involves a broad set of activites to clean, process and transform a data collection to procure meaningful insights from it.

- NumPy - create / manipulte data mathematically
- Pandas - Analise data in tabuar formate & manage presentation of data
- MAtplotlib - core library for data visualisatioin
- Seaborn - Advnce library for data visualisatioin \
- EDA - Data analysis : Exploring datasets to procure info from it and learn about data trends

What's the difference between a Python list and a NumPy array?

> NumPy offers fast, efficient ways to create and work with arrays, which are like lists that can only hold one data type, making them quicker for calculations. Unlike regular Python lists, where each item can be a different type, all items in a NumPy array are the same type. This consistency makes mathematical operations on arrays faster and more efficient.

- NumPy arrays use less memory than Python lists and allow you to specify data types, helping your code run even more efficiently.

What is an Array?

> In NumPy, an array is a core structure used to store data. Think of it as a grid or table of values, where each value is the same type, called the "array dtype." Arrays in NumPy are organized so that you can easily find and interpret each element.
>
> You can access elements in an array using various methods, like indexing with numbers, booleans, or even other arrays. Arrays can have multiple dimensions, which is called the **rank** (like rows and columns in a table), and the **shape** describes the size along each dimension.
>
> To create a NumPy array, you can start with a Python list, using nested lists for data with multiple dimensions, such as a 2D or 3D array.

- We often shorten `numpy` to `np` when importing it, like this: `import numpy as np`.
- This convention is widely used and makes code easier to read and understand.

- It's recommended to always use `np` for consistency, so others working with your code can easily follow along.

In [2]:
```
import numpy as np
```

In [2]:
```
l = [1, 2, 3, 4] # make list
l
```

Out[2]:
```
[1, 2, 3, 4]
```

In [3]:
```
a = np.array(l)  #  make array from the list
a
```

Out[3]:
```
array([1, 2, 3, 4])
```

In [4]:
```
print(a)   # calling array
```

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

In [5]:
```
np.array([])
```

Out[5]:
```
array([], dtype=float64)
```

dtype

- The desired data-type for the array. If not given, then the type willbe determined as the minimum type required to hold the objects in thesequence.

type: This is a general Python function that gives you the type of any object, like whether something is a list, integer, float, or in this case, a NumPy array. For example, calling type(arr) on a NumPy array arr would return <class 'numpy.ndarray'>.

dtype: This is specific to NumPy and stands for data type. It tells you the type of data stored in the array, such as integers, floats, or strings. For example, arr.dtype might return int32 if arr is storing 32-bit integers.

- type tells you what kind of object it is (like a NumPy array).
- dtype tells you what kind of elements are inside the array (like int32 or float64).

In [14]:
```
np.array([11,23,44] , dtype =float)
```

Out[14]:
```
array([11., 23., 44.])
```

In [15]:
```
np.array([11,23,44] , dtype =bool)

# Here True becoz , python treats Non -zero
```

Out[15]:
```
array([ True,  True,  True])
```

In [16]:
```
np.array([11,23,44] , dtype =complex)
```

Out[16]:
```
array([11.+0.j, 23.+0.j, 44.+0.j])
```

In [6]:
```
a.size  # to get no of elements in the array
```

Out[6]:
```
4
```

In [7]:
```
a.shape  # check shape of array
```

Out[7]:
```
(4,)
```

In [8]:
```
a.ndim  # cheking its dimention --- it is 1D array
```

Out[8]:
```
1
```

Numpy Arrays Vs Python Sequence

- Memory Usage:

  > **NumPy Arrays**: Use less memory because they store items in one block. This makes them faster to work with, especially with large data. ?**Python Lists**: Store items separately, which takes more memory and can be slower for big datasets.

- Speed:

  > **Array**: Much faster for math operations (like adding or multiplying numbers) because NumPy is written in a fast programming language (C).
  >
  > **List**: Slower for math operations, as Python lists aren't designed with speed in mind for big calculations.

- Data type:

  > **Array**:All items have to be the same type (like all numbers or all strings). This makes NumPy faster and more predictable.
  >
  > **List**:Items can be different types (like mixing numbers and words in one list), which makes them flexible but slower.

- Mathematical operations:

  > **Array**: You can do math on the whole array at once (like multiplying every item by 2). This is called "element-wise" operations, and it's much quicker and easier than doing each item one by one.
  >
  > **List**:You need a loop or list comprehension to perform math on each item, which takes more code and is slower.

- Multi- di data:

> **Array**:Easily handle 2D, 3D, and higher-dimensional data (like tables or grids), which is useful in data science and machine learning.
>
> **List**: Can handle multi-dimensional data, but it gets tricky and isn't as smooth as NumPy for things like tables or matrices.

In [9]:
```
a  # 1D array is a vector
```

Out[9]:
```
array([1, 2, 3, 4])
```

In [10]:
```
l2 = [[1,2,3,4],    # making 2D array from nested list
      [5,6,7,8]]
l2
```

Out[10]:
```
[[1, 2, 3, 4], [5, 6, 7, 8]]
```

In [11]:
```
ar = np.array([[45,34,12,2],[24,55,3,22]])
ar

# 2D array is an metrix
```

Out[11]:
```
array([[45, 34, 12,  2],
       [24, 55,  3, 22]])
```

In [15]:
```
b = np.array(l2)
b
```

Out[15]:
```
array([[1, 2, 3, 4],
       [5, 6, 7, 8]])
```

In [17]:
```
b.ndim #it is 2D array
# dim function gives no of dimension array having
```

Out[17]:
```
2
```

In [18]:
```
b.size #8 elemaant are present in the array
```

Out[18]:
```
8
```

In [19]:
```
b.shape # 2 rowa nd 4 columns are in the array
```

Out[19]:
```
(2, 4)
```

In [20]:
```
type(l)
```

Out[20]:
```
list
```

In [21]:
```
type(a) # defining the library numpy
```

Out[21]:
```
numpy.ndarray
```

In [22]:
```
a.dtype # it is showing that 32 bit integer
```

Out[22]:
```
dtype('int32')
```

In [13]:
```
# 3D ----  Tensor
np.array ( [[2,3,33,4,45],[23,45,56,66,2],[357,523,32,24,2],[32,32,44,33,234]])
```

Out[13]:
```
array([[  2,   3,  33,   4,  45],
       [ 23,  45,  56,  66,   2],
       [357, 523,  32,  24,   2],
       [ 32,  32,  44,  33, 234]])
```

In [24]:
```
 # All the element into array should be in same type
r = [ 3.2 , 2.3 , 4]
r
```

Out[24]:
```
[3.2, 2.3, 4]
```

In [26]:
```
r1 = np.array(r)
r1
# it is creating the 4 into float format i.e 4,0 because othe integers aare in float f
```

Out[26]:
```
array([3.2, 2.3, 4. ])
```

In [27]:
```
type(r1)
```

Out[27]:
```
numpy.ndarray
```

In [29]:
```
r1.dtype # it is showing deta type of all the elements present in the array i.e that 6
```

Out[29]:
```
dtype('float64')
```

- we cannot dtype on list
- we can only apply dtype on array because it is homogenous function

In [32]:
```
lx = np.array([[[1,2,3,4],
                [5,6,7,8]],
                [[1,3,5,7],
                [2,4,6,8]]])
lx

# here we are stacking two 2D araay together to make 3D array
```

Out[32]:
```
array([[[1, 2, 3, 4],
        [5, 6, 7, 8]],

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

In [33]:
```
lx.ndim   # it is 3D
```

Out[33]:
```
3
```

In [ ]:
```
ndarray  # n no of dimentional array can be created
```

In [35]:
```
c = np.array([[1,2,3,4],
              [2,3,4,5]])
c
```

Out[35]:
```
array([[1, 2, 3, 4],
       [2, 3, 4, 5]])
```

In [36]:
```
l = [[1,2,3,4] , [2,3,4,5]]
c = np.array(l)
c
```

Out[36]:
```
array([[1, 2, 3, 4],
       [2, 3, 4, 5]])
```

In [37]:
```
l = [[1,2,3,4] , [2,3,4,5]]
c1 = np.array(l, dtype = float)
c1
# we are passing the list into array here in float format
```

Out[37]:
```
array([[1., 2., 3., 4.],
       [2., 3., 4., 5.]])
```

In [38]:
```
l2 = [[1.3,2,3,4.2] , [2,3.1,4,5]]
c2 = np.array(l2, dtype =int    # int format
c2
```

Out[38]:
```
array([[1, 2, 3, 4],
       [2, 3, 4, 5]])
```

In [40]:
```
l2 = [[1.3,2,3,4.2] , [2,3.1,4,5]]
c3 = np.array(l2, dtype = str)
c3
# for string here it is giving asci value of unicode format in case of string in numpa
```

Out[40]:
```
array([['1.3', '2', '3', '4.2'],
       ['2', '3.1', '4', '5']], dtype='<U3')
```

- Lets take any random list i.e l = [4, 3, 'hi'] in this list acoording to storage element 4 can be stroe in 1 bit data element 3 can store 2 bit data and 'hi' is storing lets say 3 bit data here
- but in case of array ; every element into the list array stores equal storage i.e from l list every element stores equal lets say 3 3 3 bit of storage each
- numpy shows the string data in unicode formate

In [19]:
```
np.array([[1,2,'hi'],[3,4,5],['me',3,1]])
```

Out[19]:
```
array([['1', '2', 'hi'],
       ['3', '4', '5'],
       ['me', '3', '1']], dtype='<U11')
```

Zeros

- np.zeros(shape, dtype) is used to create an array that is filled entirely with zeros.
- By default, the data type (dtype) of the elements is float64, but you can specify a different type if needed, such as int.

In [21]:
```
np.zeros((3,4))  # you must have to mention it inside the tuple
```

Out[21]:
```
array([[0., 0., 0., 0.],
       [0., 0., 0., 0.],
       [0., 0., 0., 0.]])
```

In [22]:
```
np.zeros((2)) # 1D array , vector
```

Out[22]:
```
array([0., 0.])
```

In [42]:
```
l2 = np.zeros((2, 3))  # array of zeros
l2
```

Out[42]:
```
array([[0., 0., 0.],
       [0., 0., 0.]])
```

Ones

- np.ones(shape, dtype) is used to create an array where every element is set to 1.
- The default data type is also float64, but it can be changed to other types such as int or bool if needed.

In [46]:
```
l2= np.ones((2,3))  # array of ones
l2
```

Out[46]:
```
array([[1., 1., 1.],
       [1., 1., 1.]])
```

In [29]:
```
np.ones((4,8))  # array of 1 4 rows and 8 column
```

Out[29]:
```
array([[1., 1., 1., 1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1., 1., 1., 1.]])
```

In [ ]:

In [ ]:

In [49]:
```
e = np.full((2,3), 1.5)
e
# it creats the ary of all element of 1.5 of 2 rows and 3 column
```

Out[49]:
```
array([[1.5, 1.5, 1.5],
       [1.5, 1.5, 1.5]])
```

In [25]:
```
a = np.full((4,3),8)
a
```

Out[25]:
```
array([[8, 8, 8],
       [8, 8, 8],
       [8, 8, 8],
       [8, 8, 8]])
```

Random Matrix

Random function

- it will give matrix of random values
- it will be always between 0 to 1 only
- each time when you runs the cell again , it will give diff values everytime

1D random array

In [111…
```
import numpy as np
from numpy import random
```

In [112…
```
np.random.random(4)
# random array with 4 element
```

Out[112]:
```
array([0.67573228, 0.56146336, 0.76086457, 0.64991025])
```

2D random array

In [113…
```
np.random.random((2,3))
#random array of 2 rows and 3 columns
```

Out[113]:
```
array([[0.3634694 , 0.46612535, 0.25229454],
       [0.20425992, 0.34803873, 0.87625556]])
```

In [27]:
```
np.random.random((4,3))  #(4rows and 3columns)
```

Out[27]:
```
array([[0.17794175, 0.20875508, 0.23541655],
       [0.88438715, 0.0631906 , 0.20659458],
       [0.30361957, 0.6659058 , 0.96040805],
       [0.24726644, 0.80488937, 0.88338526]])
```

3D random array

In [114…
```
np.random.random((2,3,5))

# it will give 3D array
# of 2 matrix having 3 rows and 5 columns
```

Out[114]:
```
array([[[0.86591031, 0.17716431, 0.68482884, 0.6971903 , 0.67197564],
        [0.26607131, 0.46000588, 0.21535296, 0.68909712, 0.83351659],
        [0.4166723 , 0.85441736, 0.25323923, 0.81053495, 0.56945943]],

       [[0.10105012, 0.13009404, 0.88565838, 0.50866642, 0.79551451],
        [0.08978158, 0.20296468, 0.71714155, 0.24431155, 0.41395328],
        [0.64953148, 0.96376278, 0.9344305 , 0.92890963, 0.22412775]]])
```

Randint

- it will generate random **integer** no between the given range
- by default n starts from 0
- num can be repeated
- gives random numbers between given range

In [115…
```
np.random.randint(2,13)
# it will give any random no between 2 and 13
```

Out[115]:
```
2
```

In [116…
```
np.random.randint(2,10,5) # --------> ( range[start value, stop value], no of elements
#it will give any 5 random no between 2 and 10
```

Out[116]:
```
array([5, 5, 2, 3, 7])
```

In [118…
```
np.random.randint(5)   #-------> by default range will start from 0 till given no i.e
```

Out[118]:
```
1
```

Rand

- it will give **float value between 0 to 1**

In [120…
```
np.random.rand()    # it gives any rnadom float no between 0 to 1
```

Out[120]:
```
0.772604817610183
```

In [121…
```
np.random.rand(4)   #1D
```

Out[121]:
```
array([0.22343627, 0.43377475, 0.38563056, 0.55024045])
```

In [122…
```
np.random.rand(2,3)   # 2D
```

Out[122]:
```
array([[0.12199822, 0.73523421, 0.98009575],
       [0.60344112, 0.4865438 , 0.13139259]])
```

In [123…
```
np.random.rand(3,3,2)
# 3 matix, 3 rows 2 column
```

Out[123]:
```
array([[[0.03749444, 0.69994934],
        [0.91090628, 0.10770802],
        [0.74155137, 0.59147709]],

       [[0.11597925, 0.67741489],
        [0.06385999, 0.88252133],
        [0.87412051, 0.35581526]],

       [[0.95294118, 0.90267936],
        [0.78386692, 0.18538943],
        [0.96224071, 0.94047315]]])
```

Diff between Random and Rand function

- Random = you must have to pass the argument in tuple ex: np.random.random((2,3)) ----> **(double beckets)**
- Rand = you can directly pass the arguments ex: np.random.rand(2,3) ----> **single brackets**

Randn

- it gives any random no ,positive or negative also

In [125…
```
np.random.randn()  ## whenever you run this , it will give differnt differnt positive
```
Out[125]:
```
0.9266949987087295
```

In [126…
```
np.random.randn(4)
```
Out[126]:
```
array([-0.39533097,  0.93473797, -0.41430871,  0.53086712])
```

In [127…
```
np.random.randn(2,3)
```
Out[127]:
```
array([[ 1.22804506, -1.10441363,  0.13382661],
       [-1.33997085,  2.57654012, -0.32546216]])
```

In [128…
```
np.random.randn(3,2,2)

#3 matrix of 2 rows and 2 columns
```
Out[128]:
```
array([[[ 1.34509014,  1.35613754],
        [-0.19097963,  1.2536957 ]],

       [[-0.53918607, -0.83429102],
        [ 2.21677243, -0.16611044]],

       [[ 0.09675731,  1.60249978],
        [ 1.00712861,  1.11904854]]])
```

Uniform

- it returns **random float no** between given range
- the numbers **cant be repeted** here
- if range is not given and only one value is given , it wil take by default from 0 to that numer as a range
- if nothing given in the (), it will generat any random float value between 0 to 1

In [129…
```
np.random.uniform(2,3)
```
Out[129]:
```
2.5812577858766685
```

In [130…
```
np.random.uniform(2,5,10)
```
Out[130]:
```
array([3.07611742, 3.70781826, 3.1862079 , 4.61829446, 2.86583596,
       3.96372011, 4.15803616, 4.29772133, 3.86327559, 3.96197253])
```

In [131…
```
np.random.uniform(2)
```
Out[131]:
```
1.2549953472864535
```

In [132…
```
np.random.uniform()
```
Out[132]:
```
0.49983041570496933
```

Choice

- here we have to give sequence
- this funciton will give any random value from the given sequence
- if not range given , only 1 value given then it will take range from 0 to that number by default abd generate random number
- elements can be repeted here
- if you dont want rpeting element (replace = False)

In [133…
```
np.random.choice([1,3,4,5,6,9,7])
```

Out[133]:
```
7
```

In [134…
```
np.random.choice([1,3,4,5,6,9,7], 3)
# (sequence, no of element you want)
```

Out[134]:
```
array([5, 4, 6])
```

In [138…
```
np.random.choice(10,30)    #(from 10 to 30)
```

Out[138]:
```
array([5, 2, 5, 7, 8, 5, 4, 5, 2, 6, 1, 6, 9, 0, 7, 1, 1, 6, 4, 0, 1, 3,
       5, 5, 7, 7, 2, 3, 3, 1])
```

In [139…
```
np.random.choice(12)    # it take range from 0 to 12
```

Out[139]:
```
7
```

In [142…
```
np.random.choice([1,3,4,5,6,9,7], 4, replace = False)    # it will not repet the eleme
```

Out[142]:
```
array([6, 7, 9, 5])
```

In [144…
```
np.random.choice([1,3,4,5,6,9,7], 10, replace = False)



# it will throw error because , we told python not to repet the vslue and give 10 elem
# than the demanded number , it will thow error
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[144], line 1
----> 1 np.random.choice([1,3,4,5,6,9,7], 10, replace = False)

File mtrand.pyx:965, in numpy.random.mtrand.RandomState.choice()

ValueError: Cannot take a larger sample than population when 'replace=False'
```

In [145…
```
np.random.choice([1,3,4,5,6,9,7], 10, replace = True)

# but if the same code we givento python and told it to print ,
# it will print 10 elements even it is higher than range provided
# it will print it by repeating the va;ues from givn range
# because we given replace = true here
```

Out[145]:
```
array([7, 9, 1, 7, 1, 3, 4, 3, 9, 9])
```

In [ ]:

In [ ]:

arange

- arange can be called with a varying number of positional arguments
- designed to generate the sequence of numerical values like int or float
- ex :

  1] np.arange(1,10) #(sart value, end value)

  2] np.arange(1,10,2) #(start,stop,step)

Note: arange function is specificaly designed for only numerical values , it wont work with non numeric types like stirng or objects

In [50]:
```
list ( range(2,10,2))
   # range in core python
   # the step size is known but the no of elements in output is unknown
```

Out[50]:
```
[2, 4, 6, 8]
```

In [54]:
```
g = np.arange(2,10,2)   # (start, stop, step)
g
# in numpy there is arange funvtion for array
```

Out[54]:
```
array([2, 4, 6, 8])
```

In [56]:
```
# we can also write as
np.array(range(2,10,2))
# here we are typecsting the list givn into array format
```

Out[56]:
```
array([2, 4, 6, 8])
```

Linespace

- Linear space
- Linearly Seperable in given range at equal distace it creates points
- by default it will give float value

- linspace(start, end, (n) no of items you have in this range)
- linspacr will devide th ntire rang into (n) equally spaced elements
- the no of elements in known but ht estep size is not known

In [63]:
```
np.linspace(2,10,3) # divide range of 2,10 into 3 equal parts
```

Out[63]:
```
array([ 2.,  6., 10.])
```

In [32]:
```
np.linspace(1,100,10) # divide range 1 to 100 in 10 equal part
```

Out[32]:
```
array([  1.,  12.,  23.,  34.,  45.,  56.,  67.,  78.,  89., 100.])
```

In [33]:
```
np.linspace(1,100,10, dtype = int) # divide , and give integer data type
```

Out[33]:
```
array([  1,  12,  23,  34,  45,  56,  67,  78,  89, 100])
```

In [34]:
```
np.linspace(1,20,2,dtype='str')
```

Out[34]:
```
array(['1.0', '20.0'], dtype='<U32')
```

In [96]:
```
# by default it will give starting and ending point in array
np.linspace(1,10,3)
# you can see here it contain start as 1 and ending as 10 also
```

Out[96]:
```
array([ 1. ,  5.5, 10. ])
```

In [97]:
```
# if you dont want that endpoints

np.linspace(1,10,3,endpoint = False)

# here in output you wont get endpoint
```

Out[97]:
```
array([1., 4., 7.])
```

In [98]:
```
np.linspace(1,10,3,endpoint = True)   # default
```

Out[98]:
```
array([ 1. ,  5.5, 10. ])
```

In [99]:
```
np.linspace(1,10,3,retstep = True)
# it gives the differnce on which the whole range is geting devided
# here, the range of 1 to 10 is devided into 3 values with the difference of 4.5
```

Out[99]:
```
(array([ 1. ,  5.5, 10. ]), 4.5)
```

In [100…
```
np.linspace(1,10,3,retstep = False) # default it wont gives retstep
```

Out[100]:
```
array([ 1. ,  5.5, 10. ])
```

In [146…
```
# by default it will show 50 vALUE BET GIVEN RANGE
np.linspace(2,20)
```

Out[146]:
```
array([ 2.        ,  2.36734694,  2.73469388,  3.10204082,  3.46938776,
        3.83673469,  4.20408163,  4.57142857,  4.93877551,  5.30612245,
        5.67346939,  6.04081633,  6.40816327,  6.7755102 ,  7.14285714,
        7.51020408,  7.87755102,  8.24489796,  8.6122449 ,  8.97959184,
        9.34693878,  9.71428571, 10.08163265, 10.44897959, 10.81632653,
       11.18367347, 11.55102041, 11.91836735, 12.28571429, 12.65306122,
       13.02040816, 13.3877551 , 13.75510204, 14.12244898, 14.48979592,
       14.85714286, 15.2244898 , 15.59183673, 15.95918367, 16.32653061,
       16.69387755, 17.06122449, 17.42857143, 17.79591837, 18.16326531,
       18.53061224, 18.89795918, 19.26530612, 19.63265306, 20.        ])
```

In [35]:
```
np.linspace(1,5,5,dtype = 'bool')
```

Out[35]:
```
array([ True,  True,  True,  True,  True])
```

Identity Matrix

- indentity matrix is that diagonal items will be ones and evrything will be zeros

In [36]:
```
np.identity(4)
```

Out[36]:
```
array([[1., 0., 0., 0.],
       [0., 1., 0., 0.],
       [0., 0., 1., 0.],
       [0., 0., 0., 1.]])
```

In [37]:
```
np.identity(5)
```

Out[37]:
```
array([[1., 0., 0., 0., 0.],
       [0., 1., 0., 0., 0.],
       [0., 0., 1., 0., 0.],
       [0., 0., 0., 1., 0.],
       [0., 0., 0., 0., 1.]])
```

In [38]:
```
np.identity(3)
```

Out[38]:
```
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
```

In [39]:
```
np.identity(7)
```

Out[39]:
```
array([[1., 0., 0., 0., 0., 0., 0.],
       [0., 1., 0., 0., 0., 0., 0.],
       [0., 0., 1., 0., 0., 0., 0.],
       [0., 0., 0., 1., 0., 0., 0.],
       [0., 0., 0., 0., 1., 0., 0.],
       [0., 0., 0., 0., 0., 1., 0.],
       [0., 0., 0., 0., 0., 0., 1.]])
```

In [ ]:
```

```

In [ ]:
```

```

In [ ]:
```
# ML
- supervised learnng # lable
- unsupervised learning
- semi-supervised learning
- reinforcement learning
```

In [71]:
```
h = [[[1,2,3,4],[2,3,4,5],
      [3,4,5,6],[4,5,6,7]],
     [[1,2,3,4],[2,3,4,5],
      [3,4,5,6],[4,5,6,7]],
     [[1,2,3,4],[2,3,4,5],
      [3,4,5,6],[4,5,6,7]]]
ha= np.array(h)
ha
```

Out[71]:
```
array([[[1, 2, 3, 4],
        [2, 3, 4, 5],
        [3, 4, 5, 6],
        [4, 5, 6, 7]],

       [[1, 2, 3, 4],
        [2, 3, 4, 5],
        [3, 4, 5, 6],
        [4, 5, 6, 7]],

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

In [72]:
```
ha.shape
```

Out[72]:
```
(3, 4, 4)
```

reahape

- changing/ modifying the shape
- the product of arguments we are passing in reshape , should be equal to the no of items preset inside the array

In [3]:
```
np.arange(1,10).reshape(3,3)

# here , we have given command to make array from value 1 to 9 (10-1)
# it reshaping the 1D array to 2D array wih shape(3,3)
# as multiplication of shape we want (3,3) **MUST** be equal to range we provided

# range = 1 to 9
# shape we want = 3 ,3  (3*3 = 9)

# we can reshape this-
#      3 * 3 = 9
#      9 * 1 = 9
#      1 * 9 = 9

#- as we provide range till 9 , it will only work with the above shapes given
#- it wont work other than these
```

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

In [4]:
```
np.arange(1,10).reshape(9,1)
```

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

In [5]:
```
np.arange(1,10).reshape(1,9)
```

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

In [7]:
```
np.arange(1,10).reshape(2,3)

# here the example it will wont work if the product of
# argument is not equal to range we have given
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[7], line 1
----> 1 np.arange(1,10).reshape(2,3)

ValueError: cannot reshape array of size 9 into shape (2,3)
```

In [9]:
```
np.arange(1,13).reshape(2,6)
```

Out[9]:
```
array([[ 1,  2,  3,  4,  5,  6],
       [ 7,  8,  9, 10, 11, 12]])
```

In [10]:
```
np.arange(1,13).reshape(3,4)
```

Out[10]:
```
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12]])
```

In [11]:
```
np.arange(1,13).reshape(4,3)
```

Out[11]:
```
array([[ 1,  2,  3],
       [ 4,  5,  6],
       [ 7,  8,  9],
       [10, 11, 12]])
```

In [12]:
```
np.arange(1,13).reshape(6,2)
```

Out[12]:
```
array([[ 1,  2],
       [ 3,  4],
       [ 5,  6],
       [ 7,  8],
       [ 9, 10],
       [11, 12]])
```

In [13]:
```
np.arange(1,13).reshape(12,1)
```

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

In [14]:
```
np.arange(1,13).reshape(1,12)
```

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

> In above reshape exapmples we created arrays directy by giving range and shape values
>
> now lets try to reshape the existing array

In [15]:
```
h = [[[1,2,3,4],[2,3,4,5],
      [3,4,5,6],[4,5,6,7]],
     [[1,2,3,4],[2,3,4,5],
      [3,4,5,6],[4,5,6,7]],
     [[1,2,3,4],[2,3,4,5],
      [3,4,5,6],[4,5,6,7]]]
ha= np.array(h)
ha
```

Out[15]:
```
array([[[1, 2, 3, 4],
        [2, 3, 4, 5],
        [3, 4, 5, 6],
        [4, 5, 6, 7]],

       [[1, 2, 3, 4],
        [2, 3, 4, 5],
        [3, 4, 5, 6],
        [4, 5, 6, 7]],

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

In [16]:
```
ha.reshape((4,3,4))
```

Out[16]:
```
array([[[1, 2, 3, 4],
        [2, 3, 4, 5],
        [3, 4, 5, 6]],

       [[4, 5, 6, 7],
        [1, 2, 3, 4],
        [2, 3, 4, 5]],

       [[3, 4, 5, 6],
        [4, 5, 6, 7],
        [1, 2, 3, 4]],

       [[2, 3, 4, 5],
        [3, 4, 5, 6],
        [4, 5, 6, 7]]])
```

In [77]:
```
ha.reshape((2,2,3,4))
```

Out[77]:
```
array([[[[1, 2, 3, 4],
         [2, 3, 4, 5],
         [3, 4, 5, 6]],

        [[4, 5, 6, 7],
         [1, 2, 3, 4],
         [2, 3, 4, 5]]],


       [[[3, 4, 5, 6],
         [4, 5, 6, 7],
         [1, 2, 3, 4]],

        [[2, 3, 4, 5],
         [3, 4, 5, 6],
         [4, 5, 6, 7]]]])
```

In [81]:
```
ha.reshape((3,2,4,2))
```

Out[81]:
```
array([[[[1, 2],
         [3, 4],
         [2, 3],
         [4, 5]],

        [[3, 4],
         [5, 6],
         [4, 5],
         [6, 7]]],


       [[[1, 2],
         [3, 4],
         [2, 3],
         [4, 5]],

        [[3, 4],
         [5, 6],
         [4, 5],
         [6, 7]]],


       [[[1, 2],
         [3, 4],
         [2, 3],
         [4, 5]],

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

In [18]:
```
ha.reshape((2,6))
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[18], line 1
----> 1 ha.reshape((2,6,3))

ValueError: cannot reshape array of size 48 into shape (2,6,3)
```

In [90]:
```
ha.reshape((8,4))
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[90], line 1
----> 1 ha.reshape((8,4))

ValueError: cannot reshape array of size 48 into shape (8,4)
```

In [91]:
```
ha.reshape((20))
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[91], line 1
----> 1 ha.reshape((20))

ValueError: cannot reshape array of size 48 into shape (20,)
```

In [92]:
```
ha
```

Out[92]:
```
array([[[1, 2, 3, 4],
        [2, 3, 4, 5],
        [3, 4, 5, 6],
        [4, 5, 6, 7]],

       [[1, 2, 3, 4],
        [2, 3, 4, 5],
        [3, 4, 5, 6],
        [4, 5, 6, 7]],

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

In [93]:
```
ha.flatten()   # convert into 1D
```

Out[93]:
```
array([1, 2, 3, 4, 2, 3, 4, 5, 3, 4, 5, 6, 4, 5, 6, 7, 1, 2, 3, 4, 2, 3,
       4, 5, 3, 4, 5, 6, 4, 5, 6, 7, 1, 2, 3, 4, 2, 3, 4, 5, 3, 4, 5, 6,
       4, 5, 6, 7])
```

In [94]:
```
b
```

Out[94]:
```
array([[1, 2, 3, 4],
       [5, 6, 7, 8]])
```

In [96]:
```
bt = b.T    #   (dot T) It will give no of transpose
bt
```

Out[96]:
```
array([[1, 5],
       [2, 6],
       [3, 7],
       [4, 8]])
```

Identity matrix

- only digonal items will be 1 and other remaining all will be zero

In [19]:
```
np.identity(3)
```

Out[19]:
```
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
```

In [20]:
```
np.identity(6)
```

Out[20]:
```
array([[1., 0., 0., 0., 0., 0.],
       [0., 1., 0., 0., 0., 0.],
       [0., 0., 1., 0., 0., 0.],
       [0., 0., 0., 1., 0., 0.],
       [0., 0., 0., 0., 1., 0.],
       [0., 0., 0., 0., 0., 1.]])
```

In [21]:
```
np.identity(5)
```

Out[21]:
```
array([[1., 0., 0., 0., 0.],
       [0., 1., 0., 0., 0.],
       [0., 0., 1., 0., 0.],
       [0., 0., 0., 1., 0.],
       [0., 0., 0., 0., 1.]])
```

In [40]:
```
a = np.array([32,33,12])
a
```

Out[40]:
```
array([32, 33, 12])
```

In [42]:
```
type(a)
```

Out[42]:
```
numpy.ndarray
```

In [43]:
```
a.dtype
```

Out[43]:
```
dtype('int32')
```

In [44]:
```
# chaning data type

a.astype('str')
```

Out[44]:
```
array(['32', '33', '12'], dtype='<U11')
```

In [45]:
```
a.astype('bool')
```

Out[45]:
```
array([ True,  True,  True])
```

In [46]:
```
a.astype(float)
```

Out[46]:
```
array([32., 33., 12.])
```

Scaler operations on array

Arithmetic operation

- Scalar operations on Numpy arrays include performing addition or subtraction, or multiplication on each element of a Numpy array

In [48]:
```
z1 = np.array(range(1,10))
z1
```

Out[48]:
```
array([1, 2, 3, 4, 5, 6, 7, 8, 9])
```

In [56]:
```
z1 = np.arange(12).reshape(4,3)
z1
```

Out[56]:
```
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11]])
```

In [63]:
```
# addition
z1 + 34   # adding 34 in each element
```

Out[63]:
```
array([[34, 35, 36],
       [37, 38, 39],
       [40, 41, 42],
       [43, 44, 45]])
```

In [62]:
```

```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Cell In[62], line 1
----> 1 z3 = np.random.random().reshape(3,4)

AttributeError: 'float' object has no attribute 'reshape'
```

In [65]:
```
# substraction
z1 - 20
```

Out[65]:
```
array([[-20, -19, -18],
       [-17, -16, -15],
       [-14, -13, -12],
       [-11, -10,  -9]])
```

In [66]:
```
# multiplication
z1 *2
```

Out[66]:
```
array([[ 0,  2,  4],
       [ 6,  8, 10],
       [12, 14, 16],
       [18, 20, 22]])
```

In [68]:
```
# division
z1 / 2
```

Out[68]:
```
array([[0. , 0.5, 1. ],
       [1.5, 2. , 2.5],
       [3. , 3.5, 4. ],
       [4.5, 5. , 5.5]])
```

In [74]:
```
# modulo

z1 % 2
```

Out[74]:
```
array([[0, 1, 0],
       [1, 0, 1],
       [0, 1, 0],
       [1, 0, 1]], dtype=int32)
```

In [75]:
```
# floor division
z1 // 2
```

Out[75]:
```
array([[0, 0, 1],
       [1, 2, 2],
       [3, 3, 4],
       [4, 5, 5]], dtype=int32)
```

Relational operation

- The relational operators are also known as **comparison operators**
- their main function is to return either a true or false based on the value of operands.

In [76]:
```
z1
```

Out[76]:
```
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11]])
```

In [77]:
```
z1 > 30

# it checks all the elemet is greater than 30 or not
# if no it will return as false
```

Out[77]:
```
array([[False, False, False],
       [False, False, False],
       [False, False, False],
       [False, False, False]])
```

In [78]:
```
z1 < 20
```

Out[78]:
```
array([[ True,  True,  True],
       [ True,  True,  True],
       [ True,  True,  True],
       [ True,  True,  True]])
```

In [79]:
```
z1 == 30
```

Out[79]:
```
array([[False, False, False],
       [False, False, False],
       [False, False, False],
       [False, False, False]])
```

Vector Operation

- applied on both numpy array

In [80]:
```
z1
```

Out[80]:
```
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11]])
```

In [81]:
```
z2
```

Out[81]:
```
array([[12, 13, 14, 15],
       [16, 17, 18, 19],
       [20, 21, 22, 23]])
```

In [82]:
```
z1 + z2   # both arrays shape shape is not same
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[82], line 1
----> 1 z1 + z2

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

In [89]:
```
z3 = np.arange(12,24).reshape(4,3)
z3
```

Out[89]:
```
array([[12, 13, 14],
       [15, 16, 17],
       [18, 19, 20],
       [21, 22, 23]])
```

In [90]:
```
z1
```

Out[90]:
```
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11]])
```

In [91]:
```
z3
```

Out[91]:
```
array([[12, 13, 14],
       [15, 16, 17],
       [18, 19, 20],
       [21, 22, 23]])
```

In [93]:
```
z1 + z3  # both numpu arrays are having same shape
# so we can add the arrays
# itemwise addition
```

Out[93]:
```
array([[12, 14, 16],
       [18, 20, 22],
       [24, 26, 28],
       [30, 32, 34]])
```

In [95]:
```
z1*z3
```

Out[95]:
```
array([[  0,  13,  28],
       [ 45,  64,  85],
       [108, 133, 160],
       [189, 220, 253]])
```

In [96]:
```
z1 / z3
```

Out[96]:
```
array([[0.        , 0.07692308, 0.14285714],
       [0.2       , 0.25      , 0.29411765],
       [0.33333333, 0.36842105, 0.4       ],
       [0.42857143, 0.45454545, 0.47826087]])
```

In [97]:
```
z1  - z3
```

Out[97]:
```
array([[-12, -12, -12],
       [-12, -12, -12],
       [-12, -12, -12],
       [-12, -12, -12]])
```

In [98]:
```
# comparision operation on two arrays

z1 == z3
```

Out[98]:
```
array([[False, False, False],
       [False, False, False],
       [False, False, False],
       [False, False, False]])
```

In [100…
```
z1 > z3
```

Out[100]:
```
array([[False, False, False],
       [False, False, False],
       [False, False, False],
       [False, False, False]])
```

In [101…
```
z1< z3
```

Out[101]:
```
array([[ True,  True,  True],
       [ True,  True,  True],
       [ True,  True,  True],
       [ True,  True,  True]])
```

Indexing in array

- Normal way
- integer array indexing
- boolean array indexing

In [103…
```
q = [[1,2,3,4],[4,5,6,7],[6,7,8,9]]
qa = np.array(q)
qa
# 2D data
```

Out[103]:
```
array([[1, 2, 3, 4],
       [4, 5, 6, 7],
       [6, 7, 8, 9]])
```

In [104…
```
qa[0]  # 0th row
```

Out[104]:
```
array([1, 2, 3, 4])
```

In [105…
```
qa[0,2]  # coordinate index v alues
         # [row,column]
```

Out[105]:
```
3
```

In [106…
```
qa[0][3]    # normal indexing
```

Out[106]:
```
4
```

integer array indexing

Fancy Indexing

- Fancy indexing allows you to select or modify specific elements based on complex conditions or combinations of indices.
- It provides a powerful way to manipulate array data in NumPy

In [108…
```
qa[[1,2],[1,3]]    # 2D array indexing
```
Out[108]:
```
array([5, 9])
```

In [109…
```
qa[[2,0],[1,0]]
```
Out[109]:
```
array([7, 1])
```

In [110…
```
qa[[0,0],[1,2]]
```
Out[110]:
```
array([2, 3])
```

In [117…
```
qa
```
Out[117]:
```
array([[1, 2, 3, 4],
       [4, 5, 6, 7],
       [6, 7, 8, 9]])
```

In [120…
```
p3 = np.arange(8).reshape(2,2,2)
p3

# 3d consist of 2 2D array
#     array 1 = 0th
#           row 1 = 0th
#           row 2 = 1st
#     array 2 = 1st
#           row 1 = 0th
#           row 2 = 1st
```
Out[120]:
```
array([[[0, 1],
        [2, 3]],

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

In [123…
```
p3[1,1,1] #fetch 7

# :Here 3D is consists of 2 ,2D array , so Firstly we take 1 because our desired is 7
# is in second matrix which is 1 .and 2 row so 1 and second column so 1
```
Out[123]:
```
7
```

In [125…
```
p3[0,1,1]
```
Out[125]:
```
3
```

In [127…
```
p3[1,1,0]
```
Out[127]:
```
6
```

In [ ]:

Boolean array indexing

- It allows you to select elements from an array based on a Boolean condition. This allows you to extract only the elements of an array that meet a certain condition, making it easy to perform operations on specific subsets of data.

In [112…
```
qa
```

Out[112]:
```
array([[1, 2, 3, 4],
       [4, 5, 6, 7],
       [6, 7, 8, 9]])
```

In [113…
```
qa[qa % 2 == 0]
# here it will give all the coordinaates form the array which are ultiple of 2 i.e it
```

Out[113]:
```
array([2, 4, 4, 6, 6, 8])
```

In [114…
```
qa[(qa % 2 == 0) & (qa % 3 ==0)]  # we cannto use and here , we have to use bitwise &
```

Out[114]:
```
array([6, 6])
```

In [115…
```
qa[(qa % 2 == 0) | (qa % 3 ==0)] # bitwise or
```

Out[115]:
```
array([2, 3, 4, 4, 6, 6, 8, 9])
```

In [116…
```
qa[~(qa % 2 == 0)]  # except %2
```

Out[116]:
```
array([1, 3, 5, 7, 7, 9])
```

slicing

1 D slicing

In [136…
```
a = np.arange(1,10)
a
```

Out[136]:
```
array([1, 2, 3, 4, 5, 6, 7, 8, 9])
```

In [137…
```
a[2:5]
```

Out[137]:
```
array([3, 4, 5])
```

In [138…
```
a[3:5]
```

Out[138]:
```
array([4, 5])
```

In [139…
```
a[0:4:2] # [start,sstop,step]
```

Out[139]:
```
array([1, 3])
```

2 D Slicing

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

In [4]:
```
pp = np.arange(12).reshape(3,4)
pp
```

Out[4]:
```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

In [6]:
```
pp[0,:]
#Here 0 represents first row and (:) represnts Total column
```

Out[6]:
```
array([0, 1, 2, 3])
```

In [7]:
```
pp[:,2]
```

Out[7]:
```
array([ 2,  6, 10])
```

In [13]:
```
pp[1:2,1:3]

# here we sliced rows in [1:2] menas row 1only
# then we sliced column in [1:3] means col 1 and 2 (3 is excluded)
# took the common values
```

Out[13]:
```
array([[5, 6]])
```

In [14]:
```
pp[0:2,1:2]
```

Out[14]:
```
array([[1],
       [5]])
```

In [15]:
```
pp[1:2,0:3]
```

Out[15]:
```
array([[4, 5, 6]])
```

In [16]:
```
pp[0:2,0:2]
```

Out[16]:
```
array([[0, 1],
       [4, 5]])
```

In [17]:
```
pp[0:2,1:3]
```

Out[17]:
```
array([[1, 2],
       [5, 6]])
```

In [18]:
```
pp[1:3,0:1]
```

Out[18]:
```
array([[4],
       [8]])
```

In [19]:
```
pp[::2,::3]

# here we hav slices the rows in [::2] and clolumns in [::3]
# row[::2] = means from start to end all rows with step of 2
# col[::3] = means from start to end all columns with step 4
#
```

```
#      rows
#       .
#       .      0  1  2  3  -------> col
#       0 ([[ 0, 1, 2, 3],
#       1   [ 4, 5, 6, 7],
#       2   [ 8, 9, 10, 11]])
```

Out[19]:
```
array([[ 0,  3],
       [ 8, 11]])
```

In [20]:
```
pp[::2,::2]
```

Out[20]:
```
array([[ 0,  2],
       [ 8, 10]])
```

In [21]:
```
pp[1::2,0::3]
```

Out[21]:
```
array([[4, 7]])
```

In [22]:
```
pp[0::2,1::2]

# only those elements are taken which are found to be common in this
```

Out[22]:
```
array([[ 1,  3],
       [ 9, 11]])
```

In [23]:
```
pp[::2]
```

Out[23]:
```
array([[ 0,  1,  2,  3],
       [ 8,  9, 10, 11]])
```

In [25]:
```
pp[0,::2]
```

Out[25]:
```
array([0, 2])
```

In [26]:
```
pp[::,3]
```

Out[26]:
```
array([ 3,  7, 11])
```

In [27]:
```
pp[0:2,1:]
```

Out[27]:
```
array([[1, 2, 3],
       [5, 6, 7]])
```

In [28]:
```
pp[0:2,0::3]
```

Out[28]:
```
array([[0, 3],
       [4, 7]])
```

In [29]:
```
pp[1::2,1:3]
```

Out[29]:
```
array([[5, 6]])
```

In [30]:
```
pp[1:3,0::2]
```

Out[30]:
```
array([[ 4,  6],
       [ 8, 10]])
```

In [31]:
```
pp[0:2,1::2]
```

Out[31]:
```
array([[1, 3],
       [5, 7]])
```

In [119…
```
qa
```

Out[119]:
```
array([[1, 2, 3, 4],
       [4, 5, 6, 7],
       [6, 7, 8, 9]])
```

In [121…
```
qa[0:2,1:3]  # qa[rowslice, column slice ]
# here we nned 2,3,5,6 co ordintes to be sliced
# so we took rowslice from 0 t0 2 because it takes till n- 1 i.e 0 & 1
# and column slice from 1 to 3 because it takes till n -1 i.e 1 to 2
```

Out[121]:
```
array([[2, 3],
       [5, 6]])
```

In [130…
```
qa[1:,1::2]
# row wise = from 1 to last
# col wise = from 1 to last with step of 2
```

Out[130]:
```
array([[5, 7],
       [7, 9]])
```

In [131…
```
qa[::2,1::2]
```

Out[131]:
```
array([[2, 4],
       [7, 9]])
```

In [132…
```
qa
```

Out[132]:
```
array([[1, 2, 3, 4],
       [4, 5, 6, 7],
       [6, 7, 8, 9]])
```

In [134…
```
qa[0:2,1::2]
```

Out[134]:
```
array([[2, 4],
       [5, 7]])
```

Slicing 3D

In [33]:
```
d3 = np.arange(36).reshape(3,4,3)
d3
```

Out[33]:
```
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]]])
```

In [39]:
```
d3[0]
```

Out[39]:
```
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11]])
```

In [34]:
```
d3[1]
```

Out[34]:
```
array([[12, 13, 14],
       [15, 16, 17],
       [18, 19, 20],
       [21, 22, 23]])
```

In [36]:
```
d3[2]
```

Out[36]:
```
array([[24, 25, 26],
       [27, 28, 29],
       [30, 31, 32],
       [33, 34, 35]])
```

In [41]:
```
d3[0,1]
```

Out[41]:
```
array([3, 4, 5])
```

In [42]:
```
d3[0,0]
```

Out[42]:
```
array([0, 1, 2])
```

In [43]:
```
d3[0,2]
```

Out[43]:
```
array([6, 7, 8])
```

In [44]:
```
d3[0,3]
```

Out[44]:
```
array([ 9, 10, 11])
```

In [45]:
```
d3[1,0]
```

Out[45]:
```
array([12, 13, 14])
```

In [46]:
```
d3[2,0]
```

Out[46]:
```
array([24, 25, 26])
```

In [47]:
```
d3[1,1]
```

Out[47]:
```
array([15, 16, 17])
```

In [48]:
```
d3[1,2]
```

Out[48]:
```
array([18, 19, 20])
```

In [50]:
```
d3[1,3]
```

Out[50]:
```
array([21, 22, 23])
```

In [52]:
```
d3[2,1]
```

Out[52]:
```
array([27, 28, 29])
```

In [53]:
```
d3[2,2]
```

Out[53]:
```
array([30, 31, 32])
```

In [54]:
```
d3[2,3]
```

Out[54]:
```
array([33, 34, 35])
```

In [55]:
```
d3[2::]
```

Out[55]:
```
array([[[24, 25, 26],
        [27, 28, 29],
        [30, 31, 32],
        [33, 34, 35]]])
```

In [56]:
```
d3[::2]
```

Out[56]:
```
array([[[ 0,  1,  2],
        [ 3,  4,  5],
        [ 6,  7,  8],
        [ 9, 10, 11]],

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

In [58]:
```
d3

# in this 3d array , we get 3 matrix
```

Out[58]:
```
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]]])
```

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

OPration on Array

In [126…
```
qa
```

Out[126]:
```
array([[1, 2, 3, 4],
       [4, 5, 6, 7],
       [6, 7, 8, 9]])
```

In [128…
```
qa1 = qa + 1
qa1
# concatinate by adding 1
```

Out[128]:
```
array([[ 2,  3,  4,  5],
       [ 5,  6,  7,  8],
       [ 7,  8,  9, 10]])
```

In [130…
```
qa2 = qa + 1.2
qa2
```

Out[130]:
```
array([[ 2.2,  3.2,  4.2,  5.2],
       [ 5.2,  6.2,  7.2,  8.2],
       [ 7.2,  8.2,  9.2, 10.2]])
```

In [131…
```
qa ** 2  # power
```

Out[131]:
```
array([[ 1,  4,  9, 16],
       [16, 25, 36, 49],
       [36, 49, 64, 81]])
```

In [132…
```
# All arethmatic operation can be applied
```

In [133…
```
qa
```

Out[133]:
```
array([[1, 2, 3, 4],
       [4, 5, 6, 7],
       [6, 7, 8, 9]])
```

In [136…
```
qa = qa + 1
qa
# also written as qa +=1
```

Out[136]:
```
array([[ 4,  5,  6,  7],
       [ 7,  8,  9, 10],
       [ 9, 10, 11, 12]])
```

In [140…
```
l2 = np.array(l)
l2
```

Out[140]:
```
array([[1, 2, 3, 4],
       [2, 3, 4, 5]])
```

In [60]:
```
lm = ([3,6,9,2], [1,8,3,6])
l3 = np.array(lm)
l3
```

Out[60]:
```
array([[3, 6, 9, 2],
       [1, 8, 3, 6]])
```

In [145…
```
l23 = l2 + l3
l23
```

Out[145]:
```
array([[ 4,  8, 12,  6],
       [ 3, 11,  7, 11]])
```

In [146…
```
l3 % l2
```

Out[146]:
```
array([[0, 0, 0, 2],
       [1, 2, 3, 1]])
```

In [147…
```
qa
```

Out[147]:
```
array([[ 4,  5,  6,  7],
       [ 7,  8,  9, 10],
       [ 9, 10, 11, 12]])
```

In [148…
```
l2
```

Out[148]:
```
array([[1, 2, 3, 4],
       [2, 3, 4, 5]])
```

In [149…
```
l2 + qa # bwcause qa is 1D
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[149], line 1
----> 1 l2 + qa

ValueError: operands could not be broadcast together with shapes (2,4) (3,4)
```

In [152…
```
l2[0]
```

Out[152]:
```
array([1, 2, 3, 4])
```

In [151…
```
l2[0] +qa
```

Out[151]:
```
array([[ 5,  7,  9, 11],
       [ 8, 10, 12, 14],
       [10, 12, 14, 16]])
```

In [153…
```
l2 * l3 # element wise multiplication
```

Out[153]:
```
array([[ 3, 12, 27,  8],
       [ 2, 24, 12, 30]])
```

In [ ]:

In [ ]:

In [ ]:

In [ ]:

Transpose

Converts rows in to clumns ad columns into rows

In [59]:
```
pp   # take array
```

Out[59]:
```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

In [63]:
```
# method 1

np.transpose(pp)  # transpose rows into col
```

Out[63]:
```
array([[ 0,  4,  8],
       [ 1,  5,  9],
       [ 2,  6, 10],
       [ 3,  7, 11]])
```

In [65]:
```
#method 2

pp.T
```

Out[65]:
```
array([[ 0,  4,  8],
       [ 1,  5,  9],
       [ 2,  6, 10],
       [ 3,  7, 11]])
```

In [66]:
```
d3
```

Out[66]:
```
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]]])
```

In [67]:
```
d3.T
```

Out[67]:
```
array([[[ 0, 12, 24],
        [ 3, 15, 27],
        [ 6, 18, 30],
        [ 9, 21, 33]],

       [[ 1, 13, 25],
        [ 4, 16, 28],
        [ 7, 19, 31],
        [10, 22, 34]],

       [[ 2, 14, 26],
        [ 5, 17, 29],
        [ 8, 20, 32],
        [11, 23, 35]]])
```

Ravel

- Concerts any dimentionala array 1D array

In [68]:
```
pp
```

Out[68]:
```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

In [70]:
```
pp.ravel()
```

Out[70]:
```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])
```

In [71]:
```
d3
```

Out[71]:
```
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]]])
```

In [72]:
```
d3.ravel()
```

Out[72]:
```
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])
```

Stacking

- Stacking is the concept of joining arrays in NumPy.
- Arrays having the same dimensions can be stacked

In [77]:
```
a = np.arange(8).reshape(2,4)
a2 = np.arange(8,16).reshape(2,4)
print(a)
print(a2)
```

```
[[0 1 2 3]
 [4 5 6 7]]
[[ 8  9 10 11]
 [12 13 14 15]]
```

Use **hstack** for stacking horizontaly

In [81]:
```
np.hstack((a,a2))
```

Out[81]:
```
array([[ 0,  1,  2,  3,  8,  9, 10, 11],
       [ 4,  5,  6,  7, 12, 13, 14, 15]])
```

Use **vstack** for stacking verticaly

In [89]:
```
np.vstack((a,a2))
```

Out[89]:
```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15]])
```

Splitting

- opposite of stacking

- it seperates the array as per given

In [84]:
```
a
```

Out[84]:
```
array([[0, 1, 2, 3],
       [4, 5, 6, 7]])
```

In [85]:
```
a2
```

Out[85]:
```
array([[ 8,  9, 10, 11],
       [12, 13, 14, 15]])
```

hsplit for spliting horizontaly

In [91]:
```
np.hsplit(a,2)

# we have to pass here two arguments
# one - name of array
# 2nd - no you have to split the array in
```

Out[91]:
```
[array([[0, 1],
        [4, 5]]),
 array([[2, 3],
        [6, 7]])]
```

vsplit for spliting verticly

In [93]:
```
np.vsplit(a,2)  # spliting in two verticaly
```

Out[93]:
```
[array([[0, 1, 2, 3]]), array([[4, 5, 6, 7]])]
```

In [94]:
```
pp
```

Out[94]:
```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

In [96]:
```
np.hsplit(pp,2)
```

Out[96]:
```
[array([[0, 1],
        [4, 5],
        [8, 9]]),
 array([[ 2,  3],
        [ 6,  7],
        [10, 11]])]
```

In [101…
```
np.vsplit(pp,3)
```

Out[101]:
```
[array([[0, 1, 2, 3]]), array([[4, 5, 6, 7]]), array([[ 8,  9, 10, 11]])]
```

In [ ]:
```
```

Dot

In [154…
```
l_mult = l2.dot(l3)
l_mult
# dot function will give result like matrix multiplication (1st row multiply by 1st co
```

```
# but for matrix multiplication , no of rows should be equal to no of columns
# here is 2 row and 4 column
# hence it will throw error
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[154], line 1
----> 1 l_mult = l2.dot(l3)
      2 l_mult

ValueError: shapes (2,4) and (2,4) not aligned: 4 (dim 1) != 2 (dim 0)
```

In [62]:
```
l3
```

Out[62]:
```
array([[3, 6, 9, 2],
       [1, 8, 3, 6]])
```

In [61]:
```
l3.T
```

Out[61]:
```
array([[3, 1],
       [6, 8],
       [9, 3],
       [2, 6]])
```

In [156…
```
l2.dot(l3.T)
```

Out[156]:
```
array([[50, 50],
       [70, 68]])
```

In [159…
```
(l2.T).dot(l3)
```

Out[159]:
```
array([[ 5, 22, 15, 14],
       [ 9, 36, 27, 22],
       [13, 50, 39, 30],
       [17, 64, 51, 38]])
```

In [160…
```
l3.dot(l2.T)
```

Out[160]:
```
array([[50, 70],
       [50, 68]])
```

In [5]:
```
m = [[1,2,3,4],
     [5,6,7,8],
     [9,1,11,12]]
np.array(m)
```

Out[5]:
```
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9,  1, 11, 12]])
```

In [6]:
```
mm = np.array(m) +10000
mm
```

Out[6]:
```
array([[10001, 10002, 10003, 10004],
       [10005, 10006, 10007, 10008],
       [10009, 10001, 10011, 10012]])
```

array to list

In [4]:
```
import numpy as np
```

In [7]:
```
p = mm.tolist()
p
# to make list of array
```

Out[7]:
```
[[10001, 10002, 10003, 10004],
 [10005, 10006, 10007, 10008],
 [10009, 10001, 10011, 10012]]
```

In [168…
```
list(np.array(m)+ 10000)
```

Out[168]:
```
[array([10001, 10002, 10003, 10004]),
 array([10005, 10006, 10007, 10008]),
 array([10009, 10001, 10011, 10012])]
```

In [169…
```
mmm = (np.array(m) +10000).tolist()
mmm
```

Out[169]:
```
[[10001, 10002, 10003, 10004],
 [10005, 10006, 10007, 10008],
 [10009, 10001, 10011, 10012]]
```

Other mathematical operations on array

In [170…
```
 l3
```

Out[170]:
```
array([[3, 6, 9, 2],
       [1, 8, 3, 6]])
```

In [171…
```
np.sqrt(l3) # gives squareroot
```

Out[171]:
```
array([[1.73205081, 2.44948974, 3.        , 1.41421356],
       [1.        , 2.82842712, 1.73205081, 2.44948974]])
```

In [172…
```
dir(np)  # for checking the funtion under numpy
```

Out[172]:
```
['ALLOW_THREADS',
 'AxisError',
 'BUFSIZE',
 'CLIP',
 'ComplexWarning',
 'DataSource',
 'ERR_CALL',
 'ERR_DEFAULT',
 'ERR_IGNORE',
 'ERR_LOG',
 'ERR_PRINT',
 'ERR_RAISE',
 'ERR_WARN',
 'FLOATING_POINT_SUPPORT',
 'FPE_DIVIDEBYZERO',
 'FPE_INVALID',
 'FPE_OVERFLOW',
 'FPE_UNDERFLOW',
 'False_',
 'Inf',
 'Infinity',
 'MAXDIMS',
 'MAY_SHARE_BOUNDS',
 'MAY_SHARE_EXACT',
 'ModuleDeprecationWarning',
 'NAN',
 'NINF',
 'NZERO',
 'NaN',
 'PINF',
 'PZERO',
 'RAISE',
 'RankWarning',
 'SHIFT_DIVIDEBYZERO',
 'SHIFT_INVALID',
 'SHIFT_OVERFLOW',
 'SHIFT_UNDERFLOW',
 'ScalarType',
 'Tester',
 'TooHardError',
 'True_',
 'UFUNC_BUFSIZE_DEFAULT',
 'UFUNC_PYVALS_NAME',
 'VisibleDeprecationWarning',
 'WRAP',
 '_CopyMode',
 '_NoValue',
 '_UFUNC_API',
 '__NUMPY_SETUP__',
 '__all__',
 '__builtins__',
 '__cached__',
 '__config__',
 '__deprecated_attrs__',
 '__dir__',
 '__doc__',
 '__expired_functions__',
 '__file__',
 '__getattr__',
 '__git_version__',
```

```
'__loader__',
'__mkl_version__',
'__name__',
'__package__',
'__path__',
'__spec__',
'__version__',
'_add_newdoc_ufunc',
'_distributor_init',
'_financial_names',
'_globals',
'_mat',
'_pyinstaller_hooks_dir',
'_pytesttester',
'_version',
'abs',
'absolute',
'add',
'add_docstring',
'add_newdoc',
'add_newdoc_ufunc',
'all',
'allclose',
'alltrue',
'amax',
'amin',
'angle',
'any',
'append',
'apply_along_axis',
'apply_over_axes',
'arange',
'arccos',
'arccosh',
'arcsin',
'arcsinh',
'arctan',
'arctan2',
'arctanh',
'argmax',
'argmin',
'argpartition',
'argsort',
'argwhere',
'around',
'array',
'array2string',
'array_equal',
'array_equiv',
'array_repr',
'array_split',
'array_str',
'asanyarray',
'asarray',
'asarray_chkfinite',
'ascontiguousarray',
'asfarray',
'asfortranarray',
'asmatrix',
'atleast_1d',
```

```
'atleast_2d',
'atleast_3d',
'average',
'bartlett',
'base_repr',
'binary_repr',
'bincount',
'bitwise_and',
'bitwise_not',
'bitwise_or',
'bitwise_xor',
'blackman',
'block',
'bmat',
'bool8',
'bool_',
'broadcast',
'broadcast_arrays',
'broadcast_shapes',
'broadcast_to',
'busday_count',
'busday_offset',
'busdaycalendar',
'byte',
'byte_bounds',
'bytes0',
'bytes_',
'c_',
'can_cast',
'cast',
'cbrt',
'cdouble',
'ceil',
'cfloat',
'char',
'character',
'chararray',
'choose',
'clip',
'clongdouble',
'clongfloat',
'column_stack',
'common_type',
'compare_chararrays',
'compat',
'complex128',
'complex64',
'complex_',
'complexfloating',
'compress',
'concatenate',
'conj',
'conjugate',
'convolve',
'copy',
'copysign',
'copyto',
'core',
'corrcoef',
'correlate',
```

```
'cos',
'cosh',
'count_nonzero',
'cov',
'cross',
'csingle',
'ctypeslib',
'cumprod',
'cumproduct',
'cumsum',
'datetime64',
'datetime_as_string',
'datetime_data',
'deg2rad',
'degrees',
'delete',
'deprecate',
'deprecate_with_doc',
'diag',
'diag_indices',
'diag_indices_from',
'diagflat',
'diagonal',
'diff',
'digitize',
'disp',
'divide',
'divmod',
'dot',
'double',
'dsplit',
'dstack',
'dtype',
'e',
'ediff1d',
'einsum',
'einsum_path',
'emath',
'empty',
'empty_like',
'equal',
'errstate',
'euler_gamma',
'exp',
'exp2',
'expand_dims',
'expm1',
'extract',
'eye',
'fabs',
'fastCopyAndTranspose',
'fft',
'fill_diagonal',
'find_common_type',
'finfo',
'fix',
'flatiter',
'flatnonzero',
'flexible',
'flip',
```

```
'fliplr',
'flipud',
'float16',
'float32',
'float64',
'float_',
'float_power',
'floating',
'floor',
'floor_divide',
'fmax',
'fmin',
'fmod',
'format_float_positional',
'format_float_scientific',
'format_parser',
'frexp',
'from_dlpack',
'frombuffer',
'fromfile',
'fromfunction',
'fromiter',
'frompyfunc',
'fromregex',
'fromstring',
'full',
'full_like',
'gcd',
'generic',
'genfromtxt',
'geomspace',
'get_array_wrap',
'get_include',
'get_printoptions',
'getbufsize',
'geterr',
'geterrcall',
'geterrobj',
'gradient',
'greater',
'greater_equal',
'half',
'hamming',
'hanning',
'heaviside',
'histogram',
'histogram2d',
'histogram_bin_edges',
'histogramdd',
'hsplit',
'hstack',
'hypot',
'i0',
'identity',
'iinfo',
'imag',
'in1d',
'index_exp',
'indices',
'inexact',
```

```
'inf',
'info',
'infty',
'inner',
'insert',
'int0',
'int16',
'int32',
'int64',
'int8',
'int_',
'intc',
'integer',
'interp',
'intersect1d',
'intp',
'invert',
'is_busday',
'isclose',
'iscomplex',
'iscomplexobj',
'isfinite',
'isfortran',
'isin',
'isinf',
'isnan',
'isnat',
'isneginf',
'isposinf',
'isreal',
'isrealobj',
'isscalar',
'issctype',
'issubclass_',
'issubdtype',
'issubsctype',
'iterable',
'ix_',
'kaiser',
'kron',
'lcm',
'ldexp',
'left_shift',
'less',
'less_equal',
'lexsort',
'lib',
'linalg',
'linspace',
'little_endian',
'load',
'loadtxt',
'log',
'log10',
'log1p',
'log2',
'logaddexp',
'logaddexp2',
'logical_and',
'logical_not',
```

```
'logical_or',
'logical_xor',
'logspace',
'longcomplex',
'longdouble',
'longfloat',
'longlong',
'lookfor',
'ma',
'mask_indices',
'mat',
'math',
'matmul',
'matrix',
'matrixlib',
'max',
'maximum',
'maximum_sctype',
'may_share_memory',
'mean',
'median',
'memmap',
'meshgrid',
'mgrid',
'min',
'min_scalar_type',
'minimum',
'mintypecode',
'mkl',
'mod',
'modf',
'moveaxis',
'msort',
'multiply',
'nan',
'nan_to_num',
'nanargmax',
'nanargmin',
'nancumprod',
'nancumsum',
'nanmax',
'nanmean',
'nanmedian',
'nanmin',
'nanpercentile',
'nanprod',
'nanquantile',
'nanstd',
'nansum',
'nanvar',
'nbytes',
'ndarray',
'ndenumerate',
'ndim',
'ndindex',
'nditer',
'negative',
'nested_iters',
'newaxis',
'nextafter',
```

```
'nonzero',
'not_equal',
'numarray',
'number',
'obj2sctype',
'object0',
'object_',
'ogrid',
'oldnumeric',
'ones',
'ones_like',
'os',
'outer',
'packbits',
'pad',
'partition',
'percentile',
'pi',
'piecewise',
'place',
'poly',
'poly1d',
'polyadd',
'polyder',
'polydiv',
'polyfit',
'polyint',
'polymul',
'polynomial',
'polysub',
'polyval',
'positive',
'power',
'printoptions',
'prod',
'product',
'promote_types',
'ptp',
'put',
'put_along_axis',
'putmask',
'quantile',
'r_',
'rad2deg',
'radians',
'random',
'ravel',
'ravel_multi_index',
'real',
'real_if_close',
'rec',
'recarray',
'recfromcsv',
'recfromtxt',
'reciprocal',
'record',
'remainder',
'repeat',
'require',
'reshape',
```

```
'resize',
'result_type',
'right_shift',
'rint',
'roll',
'rollaxis',
'roots',
'rot90',
'round',
'round_',
'row_stack',
's_',
'safe_eval',
'save',
'savetxt',
'savez',
'savez_compressed',
'sctype2char',
'sctypeDict',
'sctypes',
'searchsorted',
'select',
'set_numeric_ops',
'set_printoptions',
'set_string_function',
'setbufsize',
'setdiff1d',
'seterr',
'seterrcall',
'seterrobj',
'setxor1d',
'shape',
'shares_memory',
'short',
'show_config',
'sign',
'signbit',
'signedinteger',
'sin',
'sinc',
'single',
'singlecomplex',
'sinh',
'size',
'sometrue',
'sort',
'sort_complex',
'source',
'spacing',
'split',
'sqrt',
'square',
'squeeze',
'stack',
'std',
'str0',
'str_',
'string_',
'subtract',
'sum',
```

```
'swapaxes',
'sys',
'take',
'take_along_axis',
'tan',
'tanh',
'tensordot',
'test',
'testing',
'tile',
'timedelta64',
'trace',
'tracemalloc_domain',
'transpose',
'trapz',
'tri',
'tril',
'tril_indices',
'tril_indices_from',
'trim_zeros',
'triu',
'triu_indices',
'triu_indices_from',
'true_divide',
'trunc',
'typecodes',
'typename',
'ubyte',
'ufunc',
'uint',
'uint0',
'uint16',
'uint32',
'uint64',
'uint8',
'uintc',
'uintp',
'ulonglong',
'unicode_',
'union1d',
'unique',
'unpackbits',
'unravel_index',
'unsignedinteger',
'unwrap',
'use_hugepage',
'ushort',
'vander',
'var',
'vdot',
'vectorize',
'version',
'void',
'void0',
'vsplit',
'vstack',
'warnings',
'where',
'who',
```

```
 'zeros',
 'zeros_like']
```

In [173…
```
len(dir(np))
```

Out[173]:
```
601
```

In [174…
```
pi
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[174], line 1
----> 1 pi

NameError: name 'pi' is not defined
```

In [175…
```
np.pi
```

Out[175]:
```
3.141592653589793
```

In [176…
```
np.sin(np.pi)
```

Out[176]:
```
1.2246467991473532e-16
```

In [177…
```
np.sin(np.pi/2)   # trignometry function
```

Out[177]:
```
1.0
```

In [178…
```
np.sin(0)
```

Out[178]:
```
0.0
```

In [179…
```
np.sqrt(9)   # squareroot functionn
```

Out[179]:
```
3.0
```

In [180…
```
np.exp
```

Out[180]:
```
<ufunc 'exp'>
```

In [181…
```
np.exp(2)   # e raise to power 2  # exponential
```

Out[181]:
```
7.38905609893065
```

In [182…
```
np.log     # logrethmic
```

Out[182]:
```
<ufunc 'log'>
```

In [183…
```
np.log(2)
```

Out[183]:
```
0.6931471805599453
```

In [184…
```
np.log10(2)
```

Out[184]:
```
0.3010299956639812
```

In [185…
```
import math as m
```

In [186…
```
m.log10(2)
```

Out[186]:
```
0.3010299956639812
```

In [188…
```
a
```

Out[188]:
```
array([1, 2, 3, 4])
```

In [189…
```
np.log(a)
```

Out[189]:
```
array([0.        , 0.69314718, 1.09861229, 1.38629436])
```

In [190…
```
np.log10(a)
```

Out[190]:
```
array([0.        , 0.30103   , 0.47712125, 0.60205999])
```

In [197…
```
np.array(q)
```

Out[197]:
```
array([[1, 2, 3, 4],
       [4, 5, 6, 7],
       [6, 7, 8, 9]])
```

In [ ]:

Sort

In [205…
```
x = np.array([[11, 12, 33, 45],
              [42, 53, 54, 71],
              [61, 91, 82, 81]])
x
```

Out[205]:
```
array([[11, 12, 33, 45],
       [42, 53, 54, 71],
       [61, 91, 82, 81]])
```

In [206…
```
x.sort(axis = 0)   # sorteed according to rows
x
```

Out[206]:
```
array([[11, 12, 33, 45],
       [42, 53, 54, 71],
       [61, 91, 82, 81]])
```

In [207…
```
x.sort(axis = 1)   # sorteed according to column
x
```

Out[207]:
```
array([[11, 12, 33, 45],
       [42, 53, 54, 71],
       [61, 81, 82, 91]])
```

In [209…
```
np.sort(e)
```

Out[209]:
```
array([[1.5, 1.5, 1.5],
       [1.5, 1.5, 1.5]])
```

In [210…
```
np.random.randn(4,4)
```

Out[210]:
```
array([[-0.26249325, -2.97182161, -0.64713835, -1.38496403],
       [ 0.44229356, -0.87929277, -0.07550638,  1.37892325],
       [-0.57119262, -0.5296497 ,  0.48306598, -1.71176944],
       [-0.8701205 , -1.19294406, -0.63424944,  1.0458619 ]])
```

In [211…
```
np.random.uniform(3,4)
```

Out[211]:
```
3.941510537524191
```

In [215…
```
np.random.uniform(3,4, (2,3))  # it will give matrix of 2 by three in which all the nc
```

Out[215]:
```
array([[3.83154218, 3.02973711, 3.7234318 ],
       [3.30574418, 3.73006232, 3.2653802 ]])
```

append

The numpy.append() appends values along the mentioned axis at the end of the array

In [9]:
```
from numpy import random
```

In [16]:
```
a = np.random.randint(20,100,10)
a
```

Out[16]:
```
array([92, 81, 43, 51, 91, 97, 96, 36, 66, 94])
```

In [17]:
```
np.append(a,100)`   # for 1D array
```

Out[17]:
```
array([ 92,  81,  43,  51,  91,  97,  96,  36,  66,  94, 100])
```

In [19]:
```
b = np.random.randn(3,2)
b
```

Out[19]:
```
array([[ 0.86354607,  1.26788122],
       [-0.54493603, -1.06821202],
       [-0.14408636, -1.15919581]])
```

In [20]:
```
np.append(b,np.ones((b.shape[0],1)))

# we want to add a new column to existing array
# for that , first we have to create that column
# we created column of ones here
```

Out[20]:
```
array([ 0.86354607,  1.26788122, -0.54493603, -1.06821202, -0.14408636,
       -1.15919581,  1.        ,  1.        ,  1.        ])
```

In [23]:
```
np.append(b,np.ones((b.shape[0],1)),axis = 1)

# now we added that column into b
```

Out[23]:
```
array([[ 0.86354607,  1.26788122,  1.        ],
       [-0.54493603, -1.06821202,  1.        ],
       [-0.14408636, -1.15919581,  1.        ]])
```

In [24]:
```
# for adding random numbers insted of ones in array
np.append(b,np.random.random((b.shape[0],1)),axis = 1)
```

Out[24]:
```
array([[ 0.86354607,  1.26788122,  0.99946684],
       [-0.54493603, -1.06821202,  0.14553826],
       [-0.14408636, -1.15919581,  0.66637408]])
```

Concatenate

- numpy.concatenate() function concatenate a sequence of arrays along an existing axis.
- Similar to stacking(hstack and vstack)

In [26]:
```
a = np.arange(6,12).reshape(2,3)
b = np.arange(12,18).reshape(2,3)
```

In [27]:
```
a
```

Out[27]:
```
array([[ 6,  7,  8],
       [ 9, 10, 11]])
```

In [29]:
```
b
```

Out[29]:
```
array([[12, 13, 14],
       [15, 16, 17]])
```

In [31]:
```
np.concatenate((a,b))   # row wise concatenation   # by default
```

Out[31]:
```
array([[ 6,  7,  8],
       [ 9, 10, 11],
       [12, 13, 14],
       [15, 16, 17]])
```

In [32]:
```
np.concatenate((a,b),axis = 0)
```

Out[32]:
```
array([[ 6,  7,  8],
       [ 9, 10, 11],
       [12, 13, 14],
       [15, 16, 17]])
```

In [33]:
```
np.concatenate((a,b), axis= 1)   # column wise concatination
```

Out[33]:
```
array([[ 6,  7,  8, 12, 13, 14],
       [ 9, 10, 11, 15, 16, 17]])
```

expand

With the help of Numpy.expand_dims() method, we can get the expanded dimensions of an array

In [48]:
```
v = np.random.randint(2,100,10)
v
```

Out[48]:
```
array([56, 70, 63, 26, 65, 44, 19, 24, 86,  2])
```

In [49]:
```
v.shape
```

Out[49]:
```
(10,)
```

In [50]:
```
# converting into 2D array
np.expand_dims(v,axis= 0)
```

Out[50]:
```
array([[56, 70, 63, 26, 65, 44, 19, 24, 86,  2]])
```

In [51]:
```
np.expand_dims(v, axis = 1)
```

Out[51]:
```
array([[56],
       [70],
       [63],
       [26],
       [65],
       [44],
       [19],
       [24],
       [86],
       [ 2]])
```

In [54]:
```
np.expand_dims(v, axis = 1).shape
```

Out[54]:
```
(10, 1)
```

where

- The numpy.where() function returns the **indices** of elements in an input array where the given condition is satisfied.

In [56]:
```
v
```

Out[56]:
```
array([56, 70, 63, 26, 65, 44, 19, 24, 86,  2])
```

In [57]:
```
np.where(v>30)
# it will find all the indices here v is greater than 30
```

Out[57]:
```
(array([0, 1, 2, 4, 5, 8], dtype=int64),)
```

In [59]:
```
np.where(v > 30 , 0, v)

# it will replace the values with 0 where value is greater than 30
# syntax = np.where(condition, if condition i true then what to print, if false what t
```

Out[59]:
```
array([ 0,  0,  0, 26,  0,  0, 19, 24,  0,  2])
```

In [60]:
```
v
```

Out[60]:
```
array([56, 70, 63, 26, 65, 44, 19, 24, 86,  2])
```

In [62]:
```
np.where(v > 30,'hi',v)   # we can also replace it with string


# if the condition saatisfies and you found the no grater than 30
```

```
# replace it with string 'hi'
# remaining keep as it is in v
```

Out[62]:
```
array(['hi', 'hi', 'hi', '26', 'hi', 'hi', '19', '24', 'hi', '2'],
      dtype='<U11')
```

In [64]:
```
np.where(v > 60,v,'hello')

# here we have given command to python that:
# if you found num > 60 in array v
# then keep it as it is in v already
# otherwise , replace it with string 'hello'


# Output Expalination:

# as v = array([56, 70, 63, 26, 65, 44, 19, 24, 86,  2])
# here all the values that are grater than 30 are kept as it is
# and those which are not are replaced with 'hello' as the given condition is not sati
```

Out[64]:
```
array(['hello', '70', '63', 'hello', '65', 'hello', 'hello', 'hello',
       '86', 'hello'], dtype='<U11')
```

Stastics

Cumsum

- numpy.cumsum() function is used when we want to compute the cumulative sum of array elements over a given axis.

In [65]:
```
v
```

Out[65]:
```
array([56, 70, 63, 26, 65, 44, 19, 24, 86,  2])
```

In [66]:
```
np.cumsum(v)
```

Out[66]:
```
array([ 56, 126, 189, 215, 280, 324, 343, 367, 453, 455])
```

In [67]:
```
a
```

Out[67]:
```
array([[ 6,  7,  8],
       [ 9, 10, 11]])
```

In [68]:
```
np.cumsum(a,axis = 1)   # row wise calculation or cumulative sum
```

Out[68]:
```
array([[ 6, 13, 21],
       [ 9, 19, 30]])
```

In [70]:
```
np.cumsum(a, axis = 0)   # columnwise calculation or cumulative sum
```

Out[70]:
```
array([[ 6,  7,  8],
       [15, 17, 19]])
```

In [71]:
```
v.cumprod()
```

```
# cumulative product
```

Out[71]:
```
array([         56,         3920,       246960,      6420960,    417362400,
       1184076416,  1022615424, -1227033600,  1849292800,  -596381696])
```

percentile

- numpy.percentile()function used to compute the nth percentile of the given data (array elements) along the specified axis.

In [72]:
```
v
```

Out[72]:
```
array([56, 70, 63, 26, 65, 44, 19, 24, 86,  2])
```

In [73]:
```
np.percentile(v,100)   # max
```

Out[73]:
```
86.0
```

In [79]:
```
np.max(v)
```

Out[79]:
```
86
```

In [74]:
```
np.percentile(v,0)   # min
```

Out[74]:
```
2.0
```

In [80]:
```
np.min(v)
```

Out[80]:
```
2
```

In [75]:
```
np.percentile(v,50)   # median
```

Out[75]:
```
50.0
```

In [76]:
```
np.median(v)
```

Out[76]:
```
50.0
```

In [77]:
```
np.mean(v)   # mean
```

Out[77]:
```
45.5
```

Flip

- The numpy.flip() function reverses the order of array elements along the specified axis, without disturbing the shape of the array

In [81]:
```
v
```

Out[81]:
```
array([56, 70, 63, 26, 65, 44, 19, 24, 86,  2])
```

In [82]:
```
np.flip(v)    # reverse
```

Out[82]:
```
array([ 2, 86, 24, 19, 44, 65, 26, 63, 70, 56])
```

In [83]:
```
a
```

Out[83]:
```
array([[ 6,  7,  8],
       [ 9, 10, 11]])
```

In [84]:
```
np.flip(a)   # for 2 D array , 3 defult fliping
```

Out[84]:
```
array([[11, 10,  9],
       [ 8,  7,  6]])
```

In [85]:
```
np.flip(a, axis = 0)   # column wise
```

Out[85]:
```
array([[ 9, 10, 11],
       [ 6,  7,  8]])
```

In [86]:
```
np.flip(a, axis = 1)    # row wise
```

Out[86]:
```
array([[ 8,  7,  6],
       [11, 10,  9]])
```

Put

- The numpy.put() function replaces specific elements of an array with given values of p_array. Array indexed works on flattened array.

In [87]:
```
v
```

Out[87]:
```
array([56, 70, 63, 26, 65, 44, 19, 24, 86,  2])
```

In [88]:
```
np.put(v,[0],[20])   # permanent chnges
```

In [89]:
```
v  # oth element is replaced by 20
```

Out[89]:
```
array([20, 70, 63, 26, 65, 44, 19, 24, 86,  2])
```

delet

- The numpy.delete() function returns a new array with the deletion of sub-arrays along with the mentioned axis.

In [90]:
```
v
```

Out[90]:
```
array([20, 70, 63, 26, 65, 44, 19, 24, 86,  2])
```

In [91]:
```
np.delete(v,0)  # delrt the value which was in index 0
```

Out[91]:
```
array([70, 63, 26, 65, 44, 19, 24, 86,  2])
```

In [92]:
```
np.delete(v,[-1,-3])  # delet index -1 nd -3 form v , we can delet multiple indices va
```

Out[92]:
```
array([20, 70, 63, 26, 65, 44, 19, 86])
```

Plotting Graph

In [109…
```
# plotting 2D plot
# x = y

x = np.linspace(-10,10,100)
x
```

Out[109]:
```
array([-10.        ,  -9.7979798 ,  -9.5959596 ,  -9.39393939,
        -9.19191919,  -8.98989899,  -8.78787879,  -8.58585859,
        -8.38383838,  -8.18181818,  -7.97979798,  -7.77777778,
        -7.57575758,  -7.37373737,  -7.17171717,  -6.96969697,
        -6.76767677,  -6.56565657,  -6.36363636,  -6.16161616,
        -5.95959596,  -5.75757576,  -5.55555556,  -5.35353535,
        -5.15151515,  -4.94949495,  -4.74747475,  -4.54545455,
        -4.34343434,  -4.14141414,  -3.93939394,  -3.73737374,
        -3.53535354,  -3.33333333,  -3.13131313,  -2.92929293,
        -2.72727273,  -2.52525253,  -2.32323232,  -2.12121212,
        -1.91919192,  -1.71717172,  -1.51515152,  -1.31313131,
        -1.11111111,  -0.90909091,  -0.70707071,  -0.50505051,
        -0.3030303 ,  -0.1010101 ,   0.1010101 ,   0.3030303 ,
         0.50505051,   0.70707071,   0.90909091,   1.11111111,
         1.31313131,   1.51515152,   1.71717172,   1.91919192,
         2.12121212,   2.32323232,   2.52525253,   2.72727273,
         2.92929293,   3.13131313,   3.33333333,   3.53535354,
         3.73737374,   3.93939394,   4.14141414,   4.34343434,
         4.54545455,   4.74747475,   4.94949495,   5.15151515,
         5.35353535,   5.55555556,   5.75757576,   5.95959596,
         6.16161616,   6.36363636,   6.56565657,   6.76767677,
         6.96969697,   7.17171717,   7.37373737,   7.57575758,
         7.77777778,   7.97979798,   8.18181818,   8.38383838,
         8.58585859,   8.78787879,   8.98989899,   9.19191919,
         9.39393939,   9.5959596 ,   9.7979798 ,  10.        ])
```

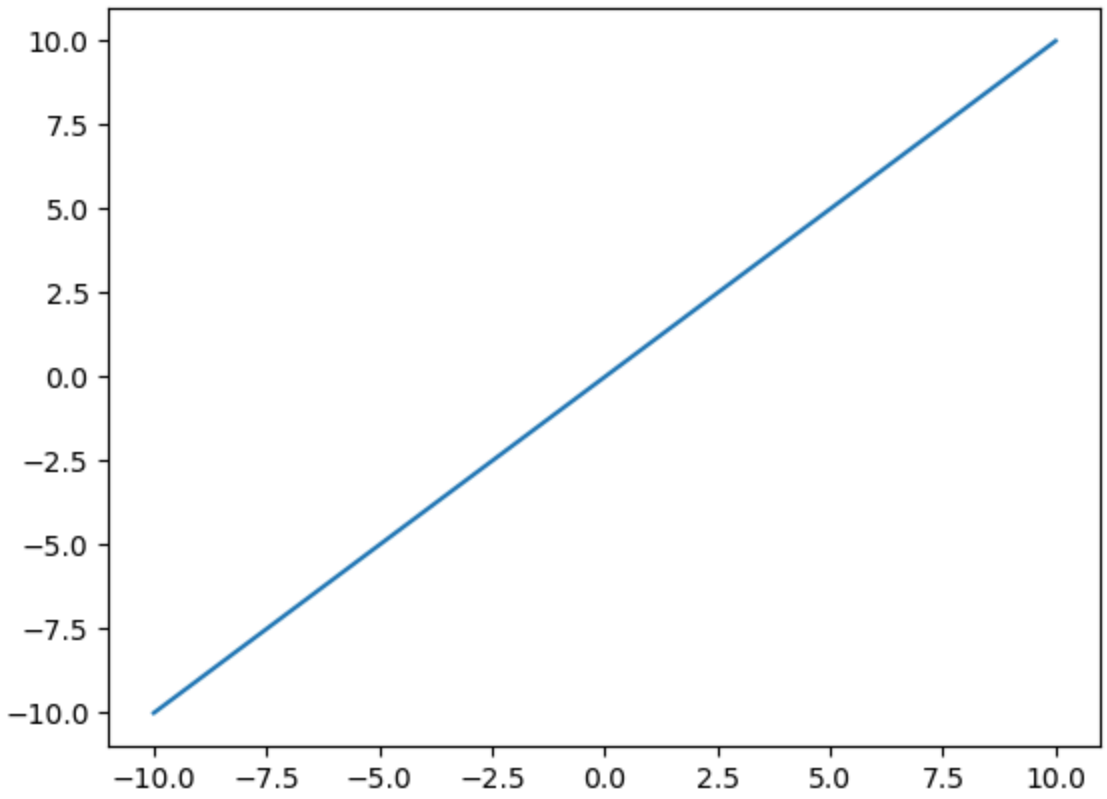
In [110…
```
# axis x = y
y = x
```

In [111…
```
y
```

Out[111]:
```
array([-10.        ,  -9.7979798 ,  -9.5959596 ,  -9.39393939,
        -9.19191919,  -8.98989899,  -8.78787879,  -8.58585859,
        -8.38383838,  -8.18181818,  -7.97979798,  -7.77777778,
        -7.57575758,  -7.37373737,  -7.17171717,  -6.96969697,
        -6.76767677,  -6.56565657,  -6.36363636,  -6.16161616,
        -5.95959596,  -5.75757576,  -5.55555556,  -5.35353535,
        -5.15151515,  -4.94949495,  -4.74747475,  -4.54545455,
        -4.34343434,  -4.14141414,  -3.93939394,  -3.73737374,
        -3.53535354,  -3.33333333,  -3.13131313,  -2.92929293,
        -2.72727273,  -2.52525253,  -2.32323232,  -2.12121212,
        -1.91919192,  -1.71717172,  -1.51515152,  -1.31313131,
        -1.11111111,  -0.90909091,  -0.70707071,  -0.50505051,
        -0.3030303 ,  -0.1010101 ,   0.1010101 ,   0.3030303 ,
         0.50505051,   0.70707071,   0.90909091,   1.11111111,
         1.31313131,   1.51515152,   1.71717172,   1.91919192,
         2.12121212,   2.32323232,   2.52525253,   2.72727273,
         2.92929293,   3.13131313,   3.33333333,   3.53535354,
         3.73737374,   3.93939394,   4.14141414,   4.34343434,
         4.54545455,   4.74747475,   4.94949495,   5.15151515,
         5.35353535,   5.55555556,   5.75757576,   5.95959596,
         6.16161616,   6.36363636,   6.56565657,   6.76767677,
         6.96969697,   7.17171717,   7.37373737,   7.57575758,
         7.77777778,   7.97979798,   8.18181818,   8.38383838,
         8.58585859,   8.78787879,   8.98989899,   9.19191919,
         9.39393939,   9.5959596 ,   9.7979798 ,  10.        ])
```

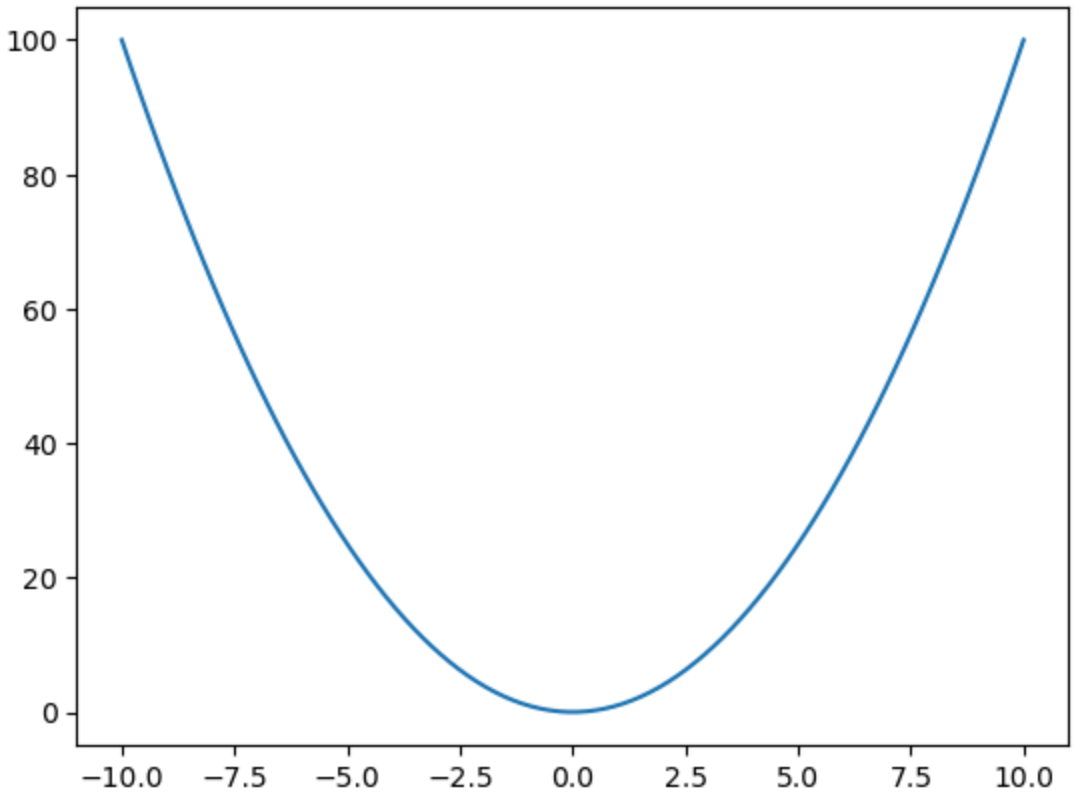
In [112…
```
# plotting the graph

import matplotlib.pyplot as plt

plt.plot(x,y)
```

Out[112]:
```
[<matplotlib.lines.Line2D at 0x21e80027a30>]
```



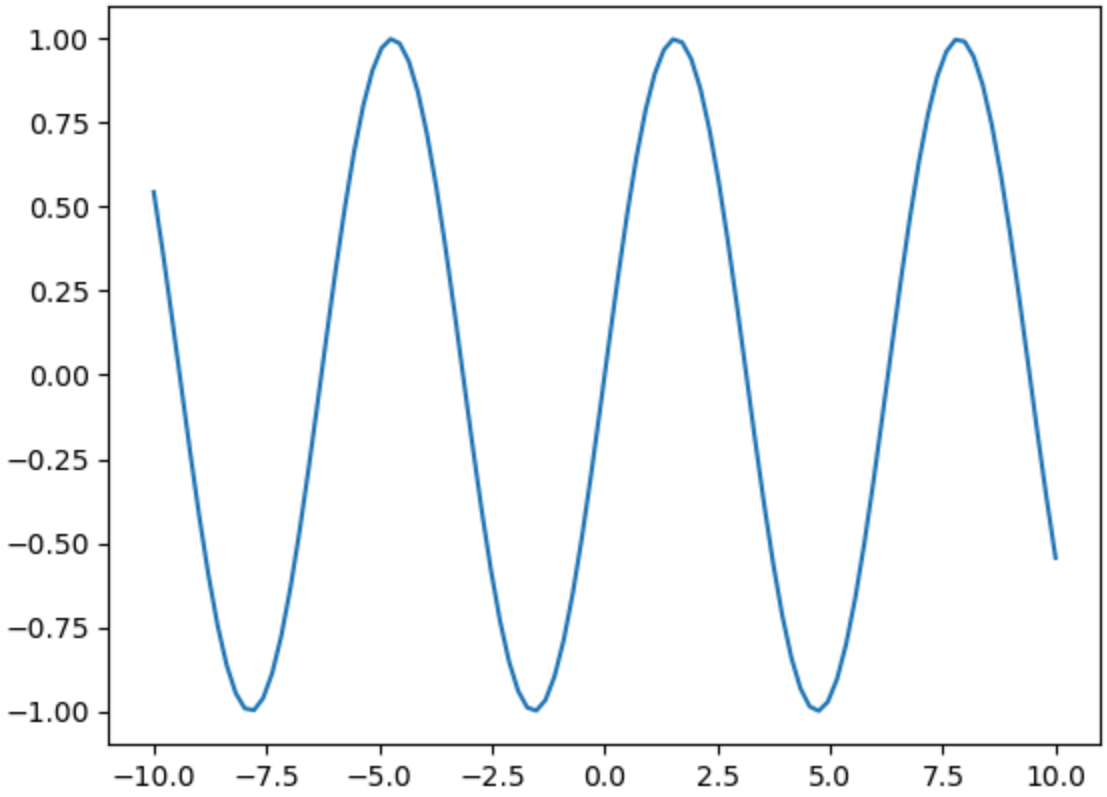
In [113…
```
y = x**2
plt.plot(x,y)
```

Out[113]:
```
[<matplotlib.lines.Line2D at 0x21e800a7ee0>]
```



In [114…
```
# sin(x)
y = np.sin(x)
plt.plot(x,y)
```

Out[114]:
```
[<matplotlib.lines.Line2D at 0x21e80272cb0>]
```



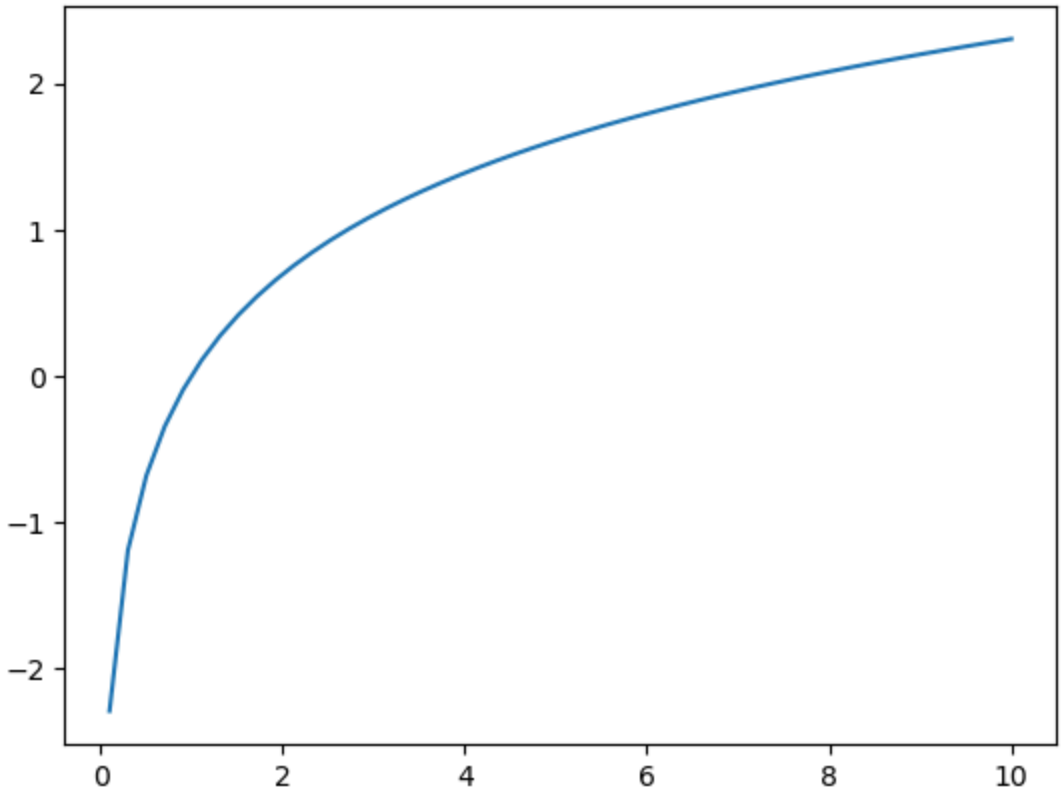
In [115…

```
# log
y = np.log(x)
plt.plot(x,y)
```

```
C:\Users\ratho\AppData\Local\Temp\ipykernel_7332\3615295649.py:2: RuntimeWarning: inv
alid value encountered in log
  y = np.log(x)
```

Out[115]:

```
[<matplotlib.lines.Line2D at 0x21e802ffa00>]
```



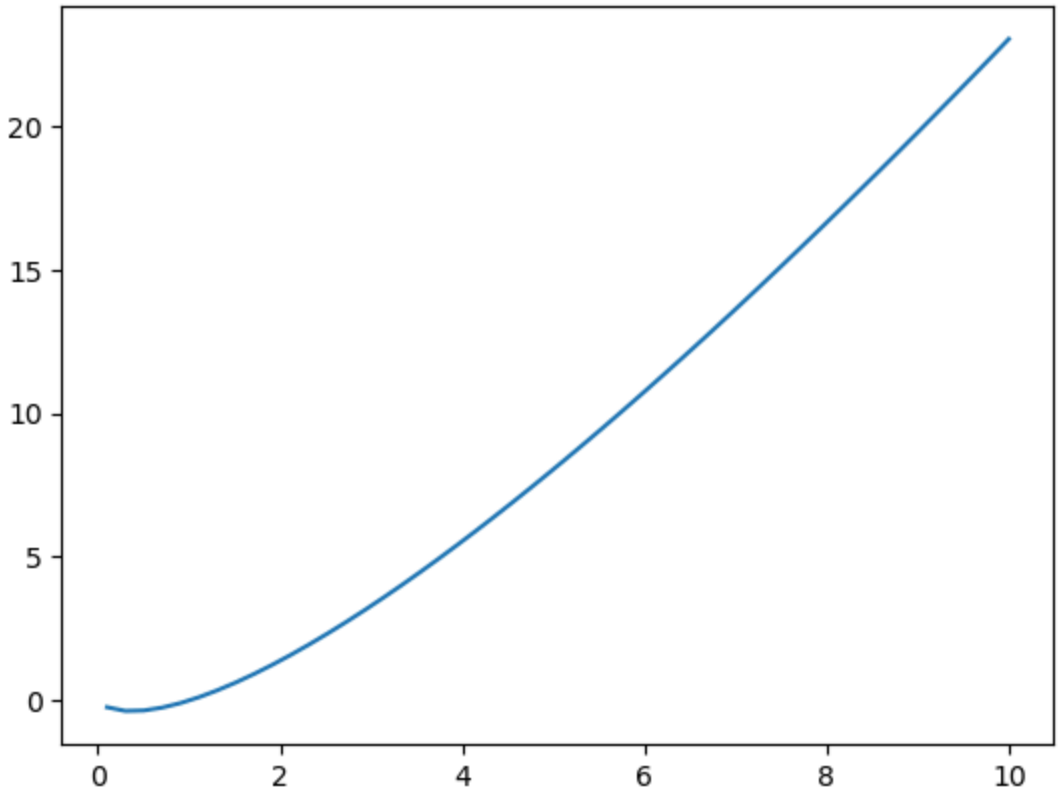
In [118…

```
y = x * np.log(x)
plt.plot(x,y)
```

```
C:\Users\ratho\AppData\Local\Temp\ipykernel_7332\793813530.py:1: RuntimeWarning: inva
lid value encountered in log
  y = x * np.log(x)
```

Out[118]:

```
[<matplotlib.lines.Line2D at 0x21e803cd300>]
```

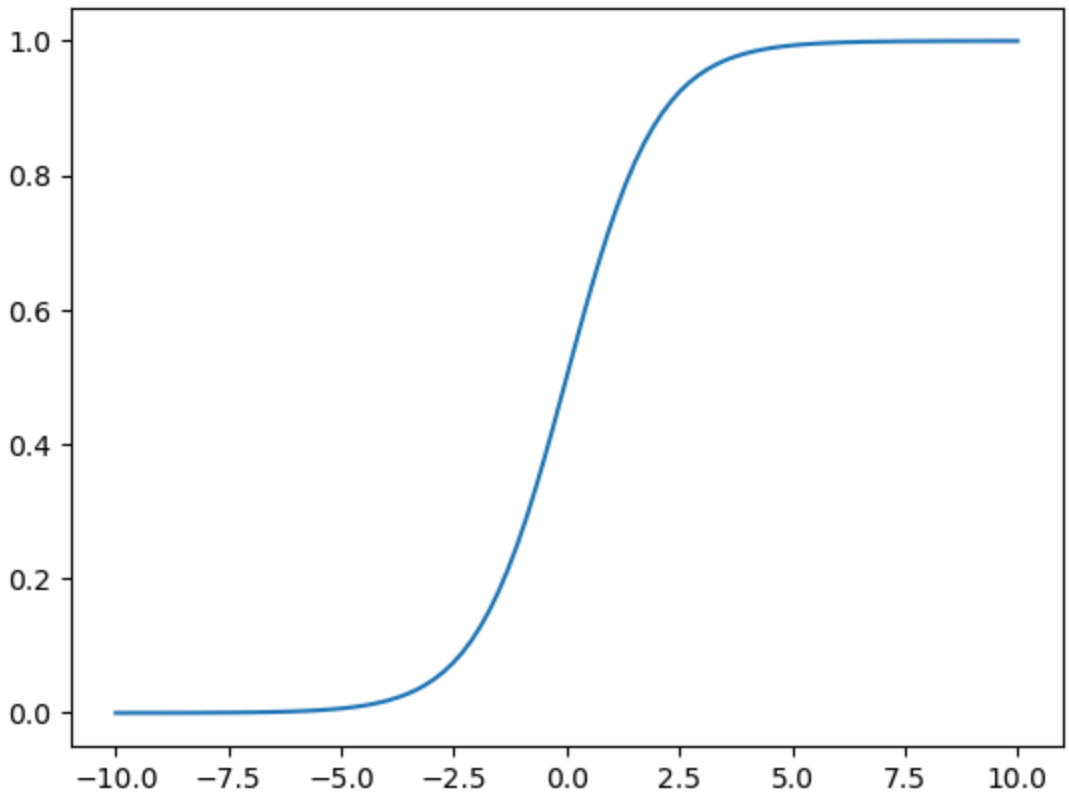


In [119…

```
y = 1/(1+np.exp(-x))
plt.plot(x,y)
```

Out[119]:

```
[<matplotlib.lines.Line2D at 0x21e804679d0>]
```



In [ ]: