## What is Numpy?

**Numpy** is the fundamental package for scietific computing in python

It is a python libarary that provides a **multidimentional array object,** various derived objects (such as masked arrays and matrices) , and an assortment of routines for fast operations on arrays, including mathematical , logical , shape manipulation , sorting , selecting , I/O, discreate Fourier transforms , basic linear algebra , basic statistical operation , random simulation and much more .

At the core of the Numpy package , is the ndarray object. this encapsulates n dimensional arrays of homogenous data types

### Creating NUmpy array

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

In [5]:
```
a = np.array([2,4,56,422,32,1])
print(a,type(a))
```

```
[  2   4  56 422  32   1] <class 'numpy.ndarray'>
```

In [6]:
```
# 2d array
new = np.array([[45,32,32,54],[232,564,64,23]])
print(new,type(new))
print("dimension of the array=",new.ndim)
print("size of the array",new.size,'in bytes')
```

```
[[ 45  32  32  54]
 [232 564  64  23]] <class 'numpy.ndarray'>
dimension of the array= 2
size of the array 8 in bytes
```

In [7]:
```
#3d array

_3darray = np.array([[[[24,25,453,564,56,14],[234,45,3,546,22,24],[5,34653,663,24
print(_3darray,type(_3darray))

print("dimension of array of array",_3darray.ndim)
```

```
print("size of the array",_3darray.size,'in bytes')
```

```
[[[    24     25    453    564     56     14]
  [   234     45      3    546     22     24]
  [     5  34653    663     24     24    254]]] <class 'numpy.ndarray'>
dimension of array of array 3
size of the array 18 in bytes
```

## 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 the sequence

In [9]:
```
np.array([11,22,33],dtype = int)
```

Out[9]:
```
array([11, 22, 33])
```

In [10]:
```
np.array([11,22,33],dtype = bool)
```

Out[10]:
```
array([ True,  True,  True])
```

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

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

In [12]:
```
np.array([11,23,44])
```

Out[12]:
```
array([11, 23, 44])
```

## Numpy arrays vs python sequences

Numpy arrays have a fixed size at creation ,unlike python lists (which can grows dynamically). changing the size of an ndarray will create a new array and delete the original.

The elements in a numpy array are all required to be of the same data type , and thus will be - the same size in memory.

Numpy arrays facilitate advanced mathematical and other types of operations on large numbers of data .Typically ,such operations are executed more efficiently and with less code than is possible using python's built in sequences .

A growing plethora of scientific and mathematical python based packages are using Numpy arrays ; though these typically support python sequences input, they convert such input to Numpy arrays prior to processing, and they often output Numpy arrays.

## arange

arange can be called with a varying number of positional arguments

In [15]:
```
np.arange(1,125) # 1 - included but 125 is not included
```

Out[15]:
```
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,  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, 121, 122, 123, 124])
```

In [16]:
```
np.arange(1,25,2)
```

Out[16]:
```
array([ 1,  3,  5,  7,  9, 11, 13, 15, 17, 19, 21, 23])
```

## reshape

Both of number products should be equal to number of items present inside the array.

In [18]:
```
j = np.arange(1,11).reshape(5,2) # converted 5 rows and 2 columns
```

In [19]:
```
j.reshape(2,5)
```

Out[19]:
```
array([[ 1,  2,  3,  4,  5],
       [ 6,  7,  8,  9, 10]])
```

In [20]:
```
np.arange(1,13).reshape(3,4) # converted 3 rows and 4 columns
```

Out[20]:
```
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12]])
```

## ones and zeros

you can initialize the values . ex in deep learning weight shape

In [22]:
```
np.ones((3,4)) # we have to mention inside tuple
```

Out[22]:
```
array([[1., 1., 1., 1.],
       [1., 1., 1., 1.],
       [1., 1., 1., 1.]])
```

In [23]:
```
np.zeros((3,4))
```

Out[23]:
```
array([[0., 0., 0., 0.],
       [0., 0., 0., 0.],
       [0., 0., 0., 0.]])
```

In [24]:
```
np.random.random((4,3))
```

Out[24]:
```
array([[0.5793949 , 0.85299877, 0.23189361],
       [0.98467846, 0.29756402, 0.02699302],
       [0.65748221, 0.62172235, 0.91620167],
       [0.43120194, 0.6184603 , 0.07552498]])
```

## linspace

it is also called as linearly space linearly separable in a given range at equal distanve it creates points.

In [26]:
```
np.linspace(-10,10,5)
```

Out[26]:
```
array([-10.,  -5.,   0.,   5.,  10.])
```

In [27]:
```
np.linspace(-2,12,6)
```

Out[27]:
```
array([-2. ,  0.8,  3.6,  6.4,  9.2, 12. ])
```

## identity

identity matrix is that diagonal items will be ones and everything will be zeros

In [29]:
```
np.identity(3)
```

Out[29]:
```
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
```

In [30]:
```
np.identity(7)
```

Out[30]:
```
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.]])
```

## array attributes

In [32]:
```
a1 = np.arange(10)
a1
```

Out[32]:
```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

In [33]:
```
a2 = np.arange(12,dtype = float).reshape(3,4)
a2
```

Out[33]:
```
array([[ 0.,  1.,  2.,  3.],
       [ 4.,  5.,  6.,  7.],
       [ 8.,  9., 10., 11.]])
```

In [34]:
```
a3 = np.arange(8) .reshape(2,2,2)
a3
```

Out[34]:
```
array([[[0, 1],
        [2, 3]],

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

## ndim

to findout given arrays number of dimensions

In [36]:
```
a1.ndim
```
Out[36]:
```
1
```
In [37]:
```
a2.ndim
```
Out[37]:
```
2
```
In [38]:
```
a3.ndim
```
Out[38]:
```
3
```

## shape

gives each item consist of no of rows and column

In [40]:
```
a1.shape
```
Out[40]:
```
(10,)
```
In [41]:
```
a2.shape
```
Out[41]:
```
(3, 4)
```
In [42]:
```
a3.shape
```
Out[42]:
```
(2, 2, 2)
```

## size

gives number of items

In [44]:
```
a3
```
Out[44]:
```
array([[[0, 1],
        [2, 3]],

       [[4, 5],
        [6, 7]]])
```
In [45]:
```
a3.size
```
Out[45]:
```
8
```
In [46]:
```
a2.size
```
Out[46]:
```
12
```
In [47]:
```
a1.size
```
Out[47]:
```
10
```

## item size

Memory occupied by the item

In [49]:
```
a1
```
Out[49]:
```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```
In [50]:
```
a1.itemsize
```
Out[50]:
```
4
```
In [51]:
```
a2.itemsize
```
Out[51]:
```
8
```
In [52]:
```
a3.itemsize
```
Out[52]:
```
4
```

## dtype

gives data type of the item

In [54]:
```
a1.dtype
```
Out[54]:
```
dtype('int32')
```
In [55]:
```
a2.dtype
```
Out[55]:
```
dtype('float64')
```
In [56]:
```
a3.dtype
```
Out[56]:
```
dtype('int32')
```

## Changing data types

In [58]:
```
x = np.array([33,22,2.5])
x.dtype
```
Out[58]:
```
dtype('float64')
```
In [59]:
```
x = x.astype(int)
x.dtype
```
Out[59]:
```
dtype('int32')
```

## Array operations

In [61]:
```
z1 = np.arange(12).reshape(3,4)
z2 = np.arange(12,24).reshape(3,4)
```
In [62]:
```
z1
```

Out[62]:
```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

In [63]:
```
z2
```

Out[63]:
```
array([[12, 13, 14, 15],
       [16, 17, 18, 19],
       [20, 21, 22, 23]])
```

### scalar operations

scaler operations on numpy arrays include performing addition or substraction or multiplication on each element of a numpy array.

In [65]:
```
z1 +2
```

Out[65]:
```
array([[ 2,  3,  4,  5],
       [ 6,  7,  8,  9],
       [10, 11, 12, 13]])
```

In [66]:
```
z1 - 2
```

Out[66]:
```
array([[-2, -1,  0,  1],
       [ 2,  3,  4,  5],
       [ 6,  7,  8,  9]])
```

In [67]:
```
z1 * 4
```

Out[67]:
```
array([[ 0,  4,  8, 12],
       [16, 20, 24, 28],
       [32, 36, 40, 44]])
```

In [68]:
```
z1 ** 2
```

Out[68]:
```
array([[  0,   1,   4,   9],
       [ 16,  25,  36,  49],
       [ 64,  81, 100, 121]])
```

In [69]:
```
z1%2
```

Out[69]:
```
array([[0, 1, 0, 1],
       [0, 1, 0, 1],
       [0, 1, 0, 1]], dtype=int32)
```

### relational operators

the relational operators also known as comparison operators ,their main function is to return either true or false based on the value of operands.

In [71]:
```
z2
```

Out[71]:
```
array([[12, 13, 14, 15],
       [16, 17, 18, 19],
       [20, 21, 22, 23]])
```

In [72]:
```
z2>2
```

Out[72]:
```
array([[ True,  True,  True,  True],
       [ True,  True,  True,  True],
       [ True,  True,  True,  True]])
```

In [73]:
```
z2>10
```

Out[73]:
```
array([[ True,  True,  True,  True],
       [ True,  True,  True,  True],
       [ True,  True,  True,  True]])
```

In [74]:
```
z2>20
```

Out[74]:
```
array([[False, False, False, False],
       [False, False, False, False],
       [False,  True,  True,  True]])
```

## Vector operations

we can apply on both numpy array

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

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

In [77]:
```
z2
```

Out[77]:
```
array([[12, 13, 14, 15],
       [16, 17, 18, 19],
       [20, 21, 22, 23]])
```

In [78]:
```
z1+z2
```

Out[78]:
```
array([[12, 14, 16, 18],
       [20, 22, 24, 26],
       [28, 30, 32, 34]])
```

In [79]:
```
z1*z2
```

Out[79]:
```
array([[  0,  13,  28,  45],
       [ 64,  85, 108, 133],
       [160, 189, 220, 253]])
```

In [80]:
```
z1-z2
```

Out[80]:
```
array([[-12, -12, -12, -12],
       [-12, -12, -12, -12],
       [-12, -12, -12, -12]])
```

In [81]:
```
z1/z2
```

Out[81]:
```
array([[0.        , 0.07692308, 0.14285714, 0.2       ],
       [0.25      , 0.29411765, 0.33333333, 0.36842105],
       [0.4       , 0.42857143, 0.45454545, 0.47826087]])
```

## Array function

In [83]:
```
k1 = np.random.random((3,3))
k1 = np.round(k1*100)
k1
```

Out[83]:
```
array([[54., 43., 93.],
       [91., 18., 11.],
       [72., 70.,  8.]])
```

In [84]:
```
np.max(k1)
```

Out[84]:
```
93.0
```

In [85]:
```
np.min(k1)
```

Out[85]:
```
8.0
```

In [86]:
```
np.sum(k1)
```

Out[86]:
```
460.0
```

In [87]:
```
np.prod(k1)
```

Out[87]:
```
156881693928960.0
```

In [88]:
```
np.max(k1,axis = 1)
```

Out[88]:
```
array([93., 91., 72.])
```

In [89]:
```
np.min(k1,axis = 0)
```

Out[89]:
```
array([54., 18.,  8.])
```

In [90]:
```
np.prod(k1,axis = 0)
```

Out[90]:
```
array([353808.,  54180.,   8184.])
```

## statistics related funnctions

In [92]:
```
k1
```

Out[92]:
```
array([[54., 43., 93.],
       [91., 18., 11.],
       [72., 70.,  8.]])
```

In [93]:
```
np.mean(k1)
```

Out[93]:
```
51.111111111111114
```

In [94]:
```
k1.mean(axis = 0)
```

Out[94]:
```
array([72.33333333, 43.66666667, 37.33333333])
```

In [95]:
```
np.median(k1)
```

Out[95]:
```
54.0
```

In [96]:
```
np.median(k1,axis =1)
```
Out[96]:
```
array([54., 18., 70.])
```
In [97]:
```
np.std(k1)
```
Out[97]:
```
31.228350525495415
```
In [98]:
```
np.var(k1)
```
Out[98]:
```
975.2098765432098
```
In [99]:
```
np.std(k1,axis = 0)
```
Out[99]:
```
array([15.10702559, 21.23414441, 39.38132665])
```

## Trignometry functions

In [101…
```
np.sin(k1)
```
Out[101…
```
array([[-0.55878905, -0.83177474, -0.94828214],
       [ 0.10598751, -0.75098725, -0.99999021],
       [ 0.25382336,  0.77389068,  0.98935825]])
```
In [102…
```
np.cos(k1)
```
Out[102…
```
array([[-0.82930983,  0.5551133 ,  0.3174287 ],
       [-0.99436746,  0.66031671,  0.0044257 ],
       [-0.96725059,  0.6333192 , -0.14550003]])
```
In [103…
```
np.tan(k1)
```
Out[103…
```
array([[ 6.73800101e-01, -1.49838734e+00, -2.98738626e+00],
       [-1.06587872e-01, -1.13731371e+00, -2.25950846e+02],
       [-2.62417378e-01,  1.22195992e+00, -6.79971146e+00]])
```

## dot product

the numpy module of python provides a function to perform the dot product of two arrays .

In [105…
```
s2 = np.arange(12).reshape(3,4)
s3 = np.arange(12,24).reshape(4,3)
```
In [106…
```
s2
```
Out[106…
```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```
In [107…
```
s3
```
Out[107…
```
array([[12, 13, 14],
       [15, 16, 17],
       [18, 19, 20],
       [21, 22, 23]])
```

In [108…
```
np.dot(s2,s3)
```

Out[108…
```
array([[114, 120, 126],
       [378, 400, 422],
       [642, 680, 718]])
```

## Log and Exponents

In [110…
```
np.exp(s2)
```

Out[110…
```
array([[1.00000000e+00, 2.71828183e+00, 7.38905610e+00, 2.00855369e+01],
       [5.45981500e+01, 1.48413159e+02, 4.03428793e+02, 1.09663316e+03],
       [2.98095799e+03, 8.10308393e+03, 2.20264658e+04, 5.98741417e+04]])
```

## Round and floor ceil

### 1.round

The numpy.round() function rounds the elements of an array to the nearest integer or to the speciffied number of decimals.

In [112…
```
# round to the nearest integer

arr = np.array([1.2,2.7,3.5,4.8])
rounded_arr = np.round(arr)
print(rounded_arr)
```

```
[1. 3. 4. 5.]
```

In [113…
```
# round to two decimals

arr = np.array([1.234, 2.567, 3.891])
rounded_arr = np.round(arr , decimals = 2)
print(rounded_arr)
```

```
[1.23 2.57 3.89]
```

In [114…
```
np.round(np.random.random((2,3))*100)
```

Out[114…
```
array([[ 29.,  87.,   8.],
       [100.,  68.,  71.]])
```

## 2.floor

The numpy.floor() function return largest integer less than or equal to each element of an array.

In [116…
```
arr = np.array([1.2,2.7,3.5,4.9])

floored_arr = np.floor(arr)
print(floored_arr)
```

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

In [117…
```
np.floor(np.random.random((2,3))*100)
```

Out[117…
```
array([[ 0., 96., 43.],
       [80., 76., 55.]])
```

### 3. ceil

The numpy.ceil() function return the smallest integer greater than or equal to each element of an array.

In [119…
```
arr = np.array([1.2,2.7,3.5,4.9])
ceiled_arr = np.ceil(arr)
print(ceiled_arr)
```
```
[2. 3. 4. 5.]
```

In [120…
```
np.ceil(np.random.random((2,3))*100)
```
Out[120…
```
array([[42.,  4., 92.],
       [15., 64., 58.]])
```

## Indexing and slicing

In [122…
```
p1 = np.arange(10)
p2 = np.arange(12).reshape(3,4)
p3 = np.arange(8).reshape(2,2,2)
```

In [123…
```
p1
```
Out[123…
```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

In [124…
```
p2
```
Out[124…
```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

In [125…
```
p3
```
Out[125…
```
array([[[0, 1],
        [2, 3]],

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

## indexing on 1d array

In [127…
```
p1
```
Out[127…
```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

In [128…
```
# fetching last item

p1[-1]
```
Out[128…
```
9
```

In [129…
```
p1[0]
```
Out[129…
```
0
```

### indexing on 2d

In [131…
```
p2
```
Out[131…
```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```
In [132…
```
p2[1,2]
```
Out[132…
```
6
```
In [133…
```
# fetching desired element :11
p2[2,-1]
```
Out[133…
```
11
```
In [134…
```
p2[2,3]
```
Out[134…
```
11
```
In [135…
```
p2[1,0]
```
Out[135…
```
4
```

### indexing on 3d (tensors)

In [137…
```
p3
```
Out[137…
```
array([[[0, 1],
        [2, 3]],

       [[4, 5],
        [6, 7]]])
```
In [138…
```
# fetching desired element :5
p3[1,0,1]
```
Out[138…
```
5
```
In [139…
```
p3[0,0,0]
```
Out[139…
```
0
```
In [140…
```
p3[1,1,1]
```
Out[140…
```
7
```

#### slicing

fetching multiple items

#### slicing on 1d

In [142…
```
p1
```

Out[142…
```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

In [143…
```
# fetching desired elements are:2,3,4

p1[2:5]
```

Out[143…
```
array([2, 3, 4])
```

In [144…
```
# alternate(same as python str


p1[2:5:2]
```

Out[144…
```
array([2, 4])
```

### slicing on 2d

In [146…
```
p2
```

Out[146…
```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

In [147…
```
p2[0,:]
```

Out[147…
```
array([0, 1, 2, 3])
```

In [148…
```
p2[:,2]
```

Out[148…
```
array([ 2,  6, 10])
```

In [149…
```
p2[1:3]
```

Out[149…
```
array([[ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

In [150…
```
p2[1:3 , 1:3]
```

Out[150…
```
array([[ 5,  6],
       [ 9, 10]])
```

In [151…
```
# fetch 0,3 and 8,11

p2
```

Out[151…
```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

In [152…
```
p2[0:3:2 , 0:4:3]
```

Out[152…
```
array([[ 0,  3],
       [ 8, 11]])
```

In [153…
```
# fetch 1,3 and 9,11

p2
```

Out[153…
```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

In [154…
```
p2[::2]# for rows
```

Out[154…
```
array([[ 0,  1,  2,  3],
       [ 8,  9, 10, 11]])
```

In [155…
```
p2[::2, 1::2] # colums
```

Out[155…
```
array([[ 1,  3],
       [ 9, 11]])
```

In [156…
```
p2
```

Out[156…
```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

In [157…
```
p2[1]
```

Out[157…
```
array([4, 5, 6, 7])
```

In [158…
```
p2[1,::3]
```

Out[158…
```
array([4, 7])
```

In [159…
```
p2
```

Out[159…
```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

In [160…
```
p2[0:2,1:]
```

Out[160…
```
array([[1, 2, 3],
       [5, 6, 7]])
```

In [161…
```
p2
```

Out[161…
```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

In [162…
```
p2[0:2]
```

Out[162…
```
array([[0, 1, 2, 3],
       [4, 5, 6, 7]])
```

In [163…
```
p2[0:2 , 1::2]
```

Out[163…
```
array([[1, 3],
       [5, 7]])
```

## slicing in 3d

In [165…
```
p3 = np.arange(27).reshape(3,3,3)
p3
```

Out[165…
```
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]]])
```

In [166…
```
# fetch second matrix
p3[1]
```

Out[166…
```
array([[ 9, 10, 11],
       [12, 13, 14],
       [15, 16, 17]])
```

In [167…
```
p3[ : :2]
```

Out[167…
```
array([[[ 0,  1,  2],
        [ 3,  4,  5],
        [ 6,  7,  8]],

       [[18, 19, 20],
        [21, 22, 23],
        [24, 25, 26]]])
```

In [168…
```
p3
```

Out[168…
```
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]]])
```

In [169…
```
p3[0]
```

Out[169…
```
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
```

In [170…
```
p3[0,1,:]
```

Out[170…
```
array([3, 4, 5])
```

In [171…
```
p3
```

Out[171...

```
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]]])
```

In [172...

```
p3[1]
```

Out[172...

```
array([[ 9, 10, 11],
       [12, 13, 14],
       [15, 16, 17]])
```

In [173...

```
p3[2]
```

Out[173...

```
array([[18, 19, 20],
       [21, 22, 23],
       [24, 25, 26]])
```

In [174...

```
p3[2,1:]
```

Out[174...

```
array([[21, 22, 23],
       [24, 25, 26]])
```

In [175...

```
p3[2,1:,1:]
```

Out[175...

```
array([[22, 23],
       [25, 26]])
```

In [176...

```
p3
```

Out[176...

```
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]]])
```

In [177...

```
p3[0::2]
```

Out[177...

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

       [[18, 19, 20],
        [21, 22, 23],
        [24, 25, 26]]])
```

In [178...

```
p3[0::2,0]
```

Out[178…
```
array([[ 0,  1,  2],
       [18, 19, 20]])
```

In [179…
```
p3[0::2,::2]
```

Out[179…
```
array([[[ 0,  1,  2],
        [ 6,  7,  8]],

       [[18, 19, 20],
        [24, 25, 26]]])
```

## Iterating

In [181…
```
p1
```

Out[181…
```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

In [182…
```
for i in p1:
    print(i)
```

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

In [183…
```
p2
```

Out[183…
```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

In [184…
```
for i in p2:
    print(i)
```

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

In [185…
```
p3
```

Out[185…
```
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]]])
```

In [186…
```
for i in p3:
```

```
    print(i)
```

```
[[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]]
```

In [187…

```
for i in np.nditer(p3):
    print(i)
```

```
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
```

## Reshaping

### Transpose ----> converts rows in columns nd columns into rows

In [189…

```
p2
```

Out[189…

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

In [190…

```
np.transpose(p2)
```

Out[190…

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

In [191…
```
p2.T
```

Out[191…
```
array([[ 0,  4,  8],
       [ 1,  5,  9],
       [ 2,  6, 10],
       [ 3,  7, 11]])
```

In [192…
```
p3
```

Out[192…
```
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]]])
```

In [193…
```
p3.T
```

Out[193…
```
array([[[ 0,  9, 18],
        [ 3, 12, 21],
        [ 6, 15, 24]],

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

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

## Ravel

converting any dimensions to 1d

In [195…
```
p2
```

Out[195…
```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

In [196…
```
p2.ravel()
```

Out[196…
```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])
```

## Stacking

stacking is the concept of joining arrays in numpy . arrays having the same dimensions can be stacked

In [198…
```
# horizontal stacking
```

```
w1 = np.arange(12).reshape(3,4)
w2 = np.arange(12,24).reshape(3,4)
```

In [199…
```
w1
```

Out[199…
```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

In [200…
```
w2
```

Out[200…
```
array([[12, 13, 14, 15],
       [16, 17, 18, 19],
       [20, 21, 22, 23]])
```

using **hstack** for horizontal stacking

In [202…
```
np.hstack((w1,w2))
```

Out[202…
```
array([[ 0,  1,  2,  3, 12, 13, 14, 15],
       [ 4,  5,  6,  7, 16, 17, 18, 19],
       [ 8,  9, 10, 11, 20, 21, 22, 23]])
```

In [203…
```
w1
```

Out[203…
```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

In [204…
```
w2
```

Out[204…
```
array([[12, 13, 14, 15],
       [16, 17, 18, 19],
       [20, 21, 22, 23]])
```

using **vstack** for vertical stacking

In [206…
```
np.vstack((w1,w2))
```

Out[206…
```
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]])
```

## splitting

its opposite of staking

In [208…
```
# horizontal splitting
w1
```

Out[208…
```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

In [209…
```
np.hsplit(w1,4) # splitting by 4
```

Out[209…
```
[array([[0],
        [4],
        [8]]),
 array([[1],
        [5],
        [9]]),
 array([[ 2],
        [ 6],
        [10]]),
 array([[ 3],
        [ 7],
        [11]])]
```

In [210…
```
w2
```

Out[210…
```
array([[12, 13, 14, 15],
       [16, 17, 18, 19],
       [20, 21, 22, 23]])
```

In [211…
```
np.vsplit(w2,3)
```

Out[211…
```
[array([[12, 13, 14, 15]]),
 array([[16, 17, 18, 19]]),
 array([[20, 21, 22, 23]])]
```

In [212…
```
# element wise addtion
import time

a = [ i for i in range(10000000)]
b = [i for i in range(10000000,20000000)]
c = []

import time

start = time.time()

for i in range(len(a)):
    c.append(a[i] +b[i])

print(time.time()-start)
```

```
2.3166346549987793
```

## numpy

In [214…
```
import numpy as np
import time
a = np.arange(10000000)
b = np.arange(10000000,20000000)

start = time.time()

c = a+b
print(time.time()-start)
```

```
0.15135598182678223
```

so **Numpy** is faster than normal python programming we can see in above

In [216...
```
2.180988073348999/0.1393413543701172
```
Out[216...
```
15.652123400177949
```

#### Memory use for list vs NUmpy

LISt

In [218...
```
import sys
R  = np.arange(10000000,dtype = np.int16)
sys.getsizeof(R)
```
Out[218...
```
20000112
```

## Advance indexing and slicing

In [220...
```
# normak indexing and slicing

w = np.arange(12).reshape(4,3)
w
```
Out[220...
```
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11]])
```
In [221...
```
# fetching 5 from array

w[1,2]
```
Out[221...
```
5
```
In [222...
```
# fetching 4,5,6,7
w[1:3]
```
Out[222...
```
array([[3, 4, 5],
       [6, 7, 8]])
```
In [223...
```
w[1:3 , 1:3]
```
Out[223...
```
array([[4, 5],
       [7, 8]])
```

### Fancy indexing

fancy indexing allows you to select or modify specific elements based on complex condition or combinations of indces .it provides a powerful way to manipulate array data in numpy

In [225...
```
w
```
Out[225...
```
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11]])
```

In [226…
```
w[[0,2,3]]
```

Out[226…
```
array([[ 0,  1,  2],
       [ 6,  7,  8],
       [ 9, 10, 11]])
```

In [227…
```
# new array

z = np.arange(24).reshape(6,4)
print(z)
```

```
[[ 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 [228…
```
z[[0,2,3,5]]
```

Out[228…
```
array([[ 0,  1,  2,  3],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15],
       [20, 21, 22, 23]])
```

In [229…
```
z[:,[0,2,3]]
```

Out[229…
```
array([[ 0,  2,  3],
       [ 4,  6,  7],
       [ 8, 10, 11],
       [12, 14, 15],
       [16, 18, 19],
       [20, 22, 23]])
```

## Boolean 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 operation on specific subsets of datat

In [231…
```
G = np.random.randint(1,100,24).reshape(6,4)
G
```

Out[231…
```
array([[74, 13, 12, 54],
       [81,  9, 83, 43],
       [43, 13, 84, 62],
       [36, 25, 36, 46],
       [42, 39, 46, 32],
       [ 3,  5, 23, 85]])
```

In [232…
```
# find all numbers greater than 50

G[G>50]
```

Out[232…
```
array([74, 54, 81, 83, 84, 62, 85])
```

In [233…
```
G>50
```

Out[233…
```
array([[ True, False, False,  True],
       [ True, False,  True, False],
       [False, False,  True,  True],
       [False, False, False, False],
       [False, False, False, False],
       [False, False, False,  True]])
```

It is best Technique to filter the data in given condition

In [235…
```
# find out even numbers
```

In [236…
```
G[G%2==0]
```

Out[236…
```
array([74, 12, 54, 84, 62, 36, 36, 46, 42, 46, 32])
```

In [237…
```
# find all numbers greater than 50 and are even

G [(G % 2 == 0) & (G>50)]
```

Out[237…
```
array([74, 54, 84, 62])
```

In [238…
```
G%7==0
```

Out[238…
```
array([[False, False, False, False],
       [False, False, False, False],
       [False, False,  True, False],
       [False, False, False, False],
       [ True, False, False, False],
       [False, False, False, False]])
```

In [239…
```
G[~(G%7==0)]
```

Out[239…
```
array([74, 13, 12, 54, 81,  9, 83, 43, 43, 13, 62, 36, 25, 36, 46, 39, 46,
       32,  3,  5, 23, 85])
```

## Broadcasting

- used in vectorization
- the term broadcasting describes how numpy treats arrays with different shapes during arithmetic operations.
- The smaller array is "broadcast" across the larger array so that they have compatible shapes

In [241…
```
a = np.arange(6).reshape(2,3)
b= np.arange(6,12).reshape(2,3)
print(a)
print(b)

print(a+b)
```

```
[[0 1 2]
 [3 4 5]]
[[ 6  7  8]
 [ 9 10 11]]
[[ 6  8 10]
 [12 14 16]]
```

In [242...
```python
a = np.arange(6).reshape(2,3)
b = np.arange(3).reshape(1,3)

print(a)
print(b)

print(a+b)
```

```
[[0 1 2]
 [3 4 5]]
[[0 1 2]]
[[0 2 4]
 [3 5 7]]
```

In [243...
```python
np.vstack(a)
```

Out[243...
```
array([[0, 1, 2],
       [3, 4, 5]])
```

In [244...
```python
b
```

Out[244...
```
array([[0, 1, 2]])
```

In [245...
```python
a+b
```

Out[245...
```
array([[0, 2, 4],
       [3, 5, 7]])
```

### Broadcasting rules

1. Make the two arrays have the same number of dimensions.

- if the numbes of dimensions of the two arrays are differernt , add new dimensions with size 1 to head of the array with the smaller dimension.

  Ex: (3,2) = 2d , (3) = 1d ----> convert into (1,3) (3,3,3) = 3d ,(3) = 1d --------> convert into (1,1,3)

  B. Make each dimension of the two arrays the same size.

  - if the sizes of each dimensions of the two arrays do not match ,dimension with size 1 are streched to the size of the other array.
  - ex : (3,3) = 2d ,(3) = 1d ----> converted (1,3) than strech to (3,3)

  C. if there is a dimension whose size is not 1 in either of two arrays , it cannot be broadcasted and an error is raised

In [247...
```
a= np.arange (12).reshape(4,3)
b = np.arange(3)
```

In [248...
```
print(a)
```

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

In [249...
```
print(b)
```

```
[0 1 2]
```

In [250...
```
print(a+b)
```

```
[[ 0  2  4]
 [ 3  5  7]
 [ 6  8 10]
 [ 9 11 13]]
```

Explanation: Arithmetic operation possible because , here a = (4,3) is 2d and b = (3) is 1d so did converted (3) to (1,3) and streched to (4,3)

In [252...
```
# could not broadcast

a = np.arange(12).reshape(3,4)
b= np.arange(3)

print(a)

print("-------------------------------------------------------")
print(b)
```

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

In [253...
```
print(a+b)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[253], line 1
----> 1 print(a+b)

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

Explanation : Arithmetic operation not possible because , Here a = (3,4) is 2d and b =(3) is 1d so did converted (3) to (1,3) and streched to (3,3) but a , is not equls to b.so it got failed

In [261...
```
a = np.arange(3).reshape(1,3)
b = np.arange(3).reshape(3,1)

print(a)
```

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

In [263…
```
print(b)
```
```
[[0]
 [1]
 [2]]
```

In [265…
```
print(a+b)
```
```
[[0 1 2]
 [1 2 3]
 [2 3 4]]
```

Explanation :Arithmetic Operation possible because ,Here a = (1,3) is 2D and b = (3,1) is 2d soc did converted(1,3) to (3,3) and b(3,1) convert (1) to 3 than (3,3) finally it equally.

In [268…
```
a = np.arange (3).reshape(1,3)
b = np.arange(4).reshape(4,1)
print(a)
print(b)

print(a+b)
```
```
[[0 1 2]]
[[0]
 [1]
 [2]
 [3]]
[[0 1 2]
 [1 2 3]
 [2 3 4]
 [3 4 5]]
```

In [270…
```
a = np.array([1])
# shape -> (1,1) streched to 2,2
b= np.arange (4).reshape(2,2)

# shape - > (2,2)

print(a)
print(b)
print(a+b)
```
```
[1]
[[0 1]
 [2 3]]
[[1 2]
 [3 4]]
```

In [275…
```
a = np.arange(12).reshape(3,4)
b = np.arange(12).reshape(4,3)

print(a)
print(b)


print(a+b)
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
[[ 0  1  2]
 [ 3  4  5]
 [ 6  7  8]
 [ 9 10 11]]
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[275], line 8
      4 print(a)
      5 print(b)
----> 8 print(a+b)

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

In [277…

```
a = np.arange(16).reshape(4,4)
b = np.arange(4).reshape(2,2)

print(a)
print(b)

print(a+b)
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]
 [12 13 14 15]]
[[0 1]
 [2 3]]
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[277], line 7
      4 print(a)
      5 print(b)
----> 7 print(a+b)

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

In [ ]:

## Working with mathematical formulas

In [280…

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

Out[280…

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

In [282…

```
np.sum(k)
```

Out[282…

```
45
```

In [284…

```
np.sin(k)
```

Out[284…

```
array([ 0.        ,  0.84147098,  0.90929743,  0.14112001, -0.7568025 ,
       -0.95892427, -0.2794155 ,  0.6569866 ,  0.98935825,  0.41211849])
```

## sigmoid

In [287…
```python
def sigmoid(array):
    return 1/(1+np.exp(-(array)))
k = np.arange(10)
sigmoid(k)
```

Out[287…
```
array([0.5       , 0.73105858, 0.88079708, 0.95257413, 0.98201379,
       0.99330715, 0.99752738, 0.99908895, 0.99966465, 0.99987661])
```

In [289…
```python
k = np.arange(100)
sigmoid(k)
```

Out[289…
```
array([0.5       , 0.73105858, 0.88079708, 0.95257413, 0.98201379,
       0.99330715, 0.99752738, 0.99908895, 0.99966465, 0.99987661,
       0.9999546 , 0.9999833 , 0.99999386, 0.99999774, 0.99999917,
       0.99999969, 0.99999989, 0.99999996, 0.99999998, 0.99999999,
       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.        , 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.        , 1.        ,
       1.        , 1.        , 1.        , 1.        , 1.        ,
       1.        , 1.        , 1.        , 1.        , 1.        ,
       1.        , 1.        , 1.        , 1.        , 1.        ])
```

## mean squared error

In [292…
```python
actual = np.random.randint(1,50,25)
predicted = np.random.randint(1,50,25)
```

In [294…
```python
actual
```

Out[294…
```
array([ 9, 39, 26,  9, 23, 15,  1, 16,  2,  4, 34, 43, 32, 26, 30, 36, 10,
       35, 27, 35, 36, 20, 20, 35, 13])
```

In [296…
```python
predicted
```

Out[296…
```
array([39, 21, 30, 16, 11, 43, 38,  2, 36, 18, 33, 47, 40, 29, 27, 27, 49,
       28, 39,  1,  5, 12,  5, 25,  8])
```

In [298…
```python
actual == predicted
```

Out[298…
```
array([False, False, False, False, False, False, False, False, False,
       False, False, False, False, False, False, False, False, False,
       False, False, False, False, False, False, False])
```

In [300…
```python
actual is predicted
```

Out[300… False

In [306…
```
def mse(actual , predicted):
    print(np.mean((actual - predicted)**2))

mse(actual,predicted)
```
```
382.36
```

In [308…
```
actual - predicted
```
Out[308…
```
array([-30,  18,  -4,  -7,  12, -28, -37,  14, -34, -14,   1,  -4,  -8,
        -3,   3,   9, -39,   7, -12,  34,  31,   8,  15,  10,   5])
```

In [310…
```
(actual - predicted)**2
```
Out[310…
```
array([ 900,  324,   16,   49,  144,  784, 1369,  196, 1156,  196,    1,
         16,   64,    9,    9,   81, 1521,   49,  144, 1156,  961,   64,
        225,  100,   25])
```

## Working with missing values

In [313…
```
s = np.array([1,2,3,4,np.nan,6])
print(s)
```
```
[ 1.  2.  3.  4. nan  6.]
```

In [323…
```
np.isnan(s)
```
Out[323…
```
array([False, False, False, False,  True, False])
```

In [325…
```
s[np.isnan(s)]# nan values
```
Out[325…
```
array([nan])
```

In [327…
```
s[~np.isnan(s)]
```
Out[327…
```
array([1., 2., 3., 4., 6.])
```

## Plotting Graphs

In [330…
```
# plotting a 2d plot
#x = y

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

Out[330…

```
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 [332…

```
y = x
```

In [334…

```
y
```

Out[334…

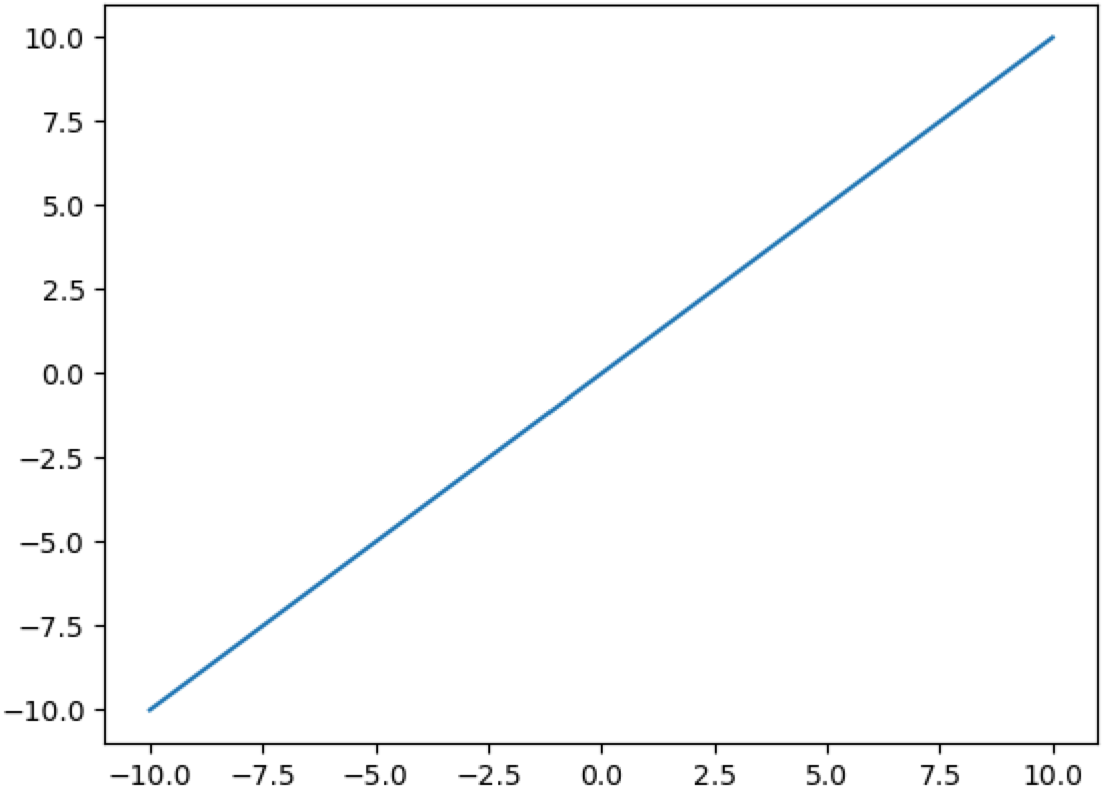
```
array([-10.        ,  -9.7979798 ,  -9.5959596 ,  -9.39393939,
        -9.19191919,  -8.98989899,  -8.78787879,  -8.58585859,
        -8.38383838,  -8.18181818,  -7.97979798,  -7.77777778,
        -7.57575758,  -7.37373737,  -7.17171717,  -6.96969697,
        -6.76767677,  -6.56565657,  -6.36363636,  -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 [336…

```
from matplotlib import pyplot as plt

plt.plot(x,y)
```

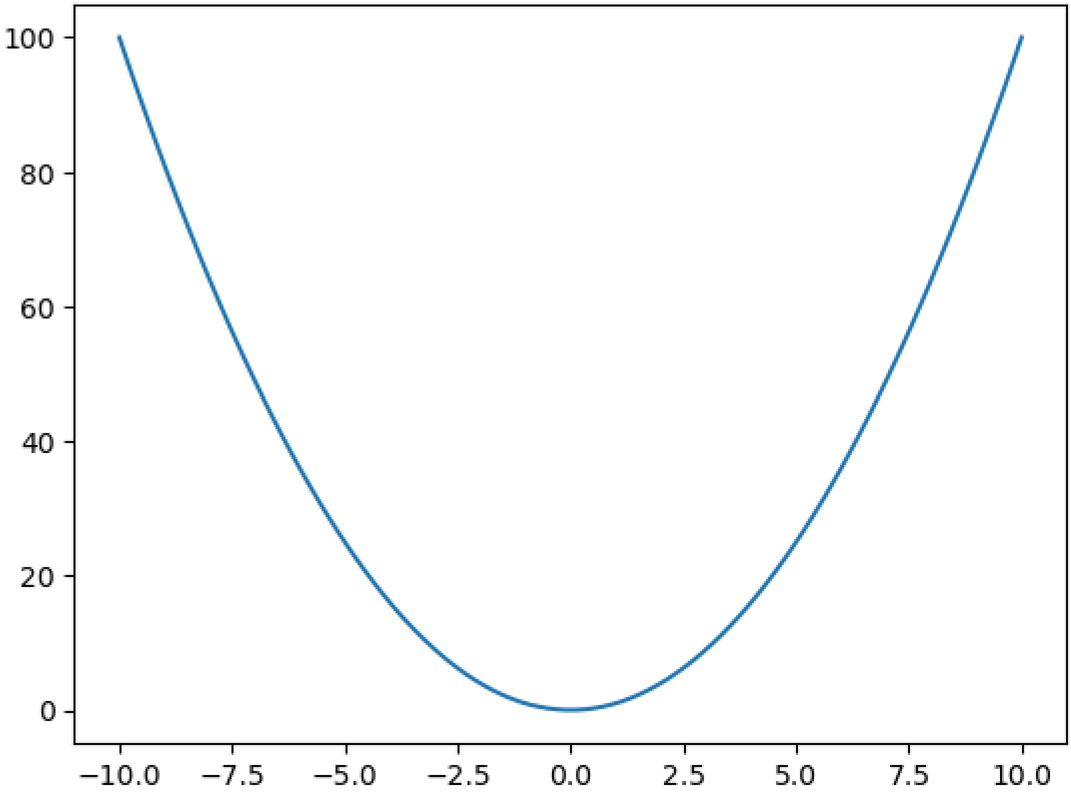
Out[336…

```
[<matplotlib.lines.Line2D at 0x26e0279b8d0>]
```



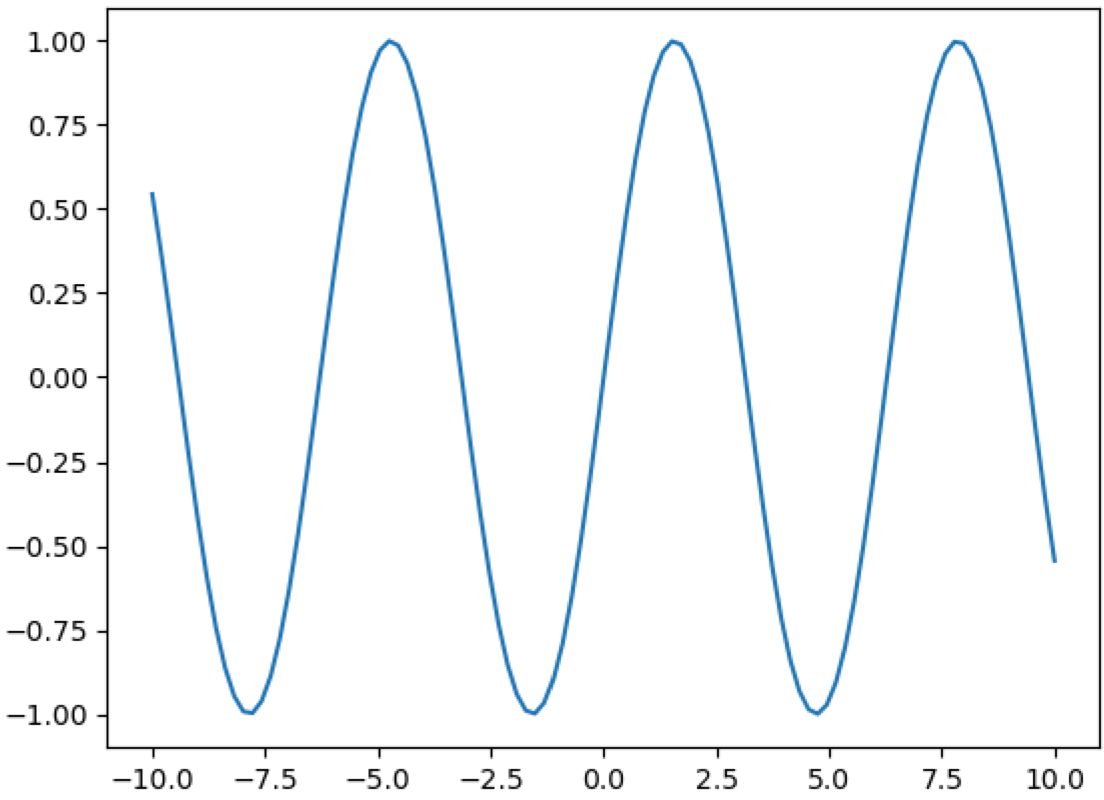

In [340…

```
y = x**2

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

plt.plot(x,y)
plt.show()
```



In [342…

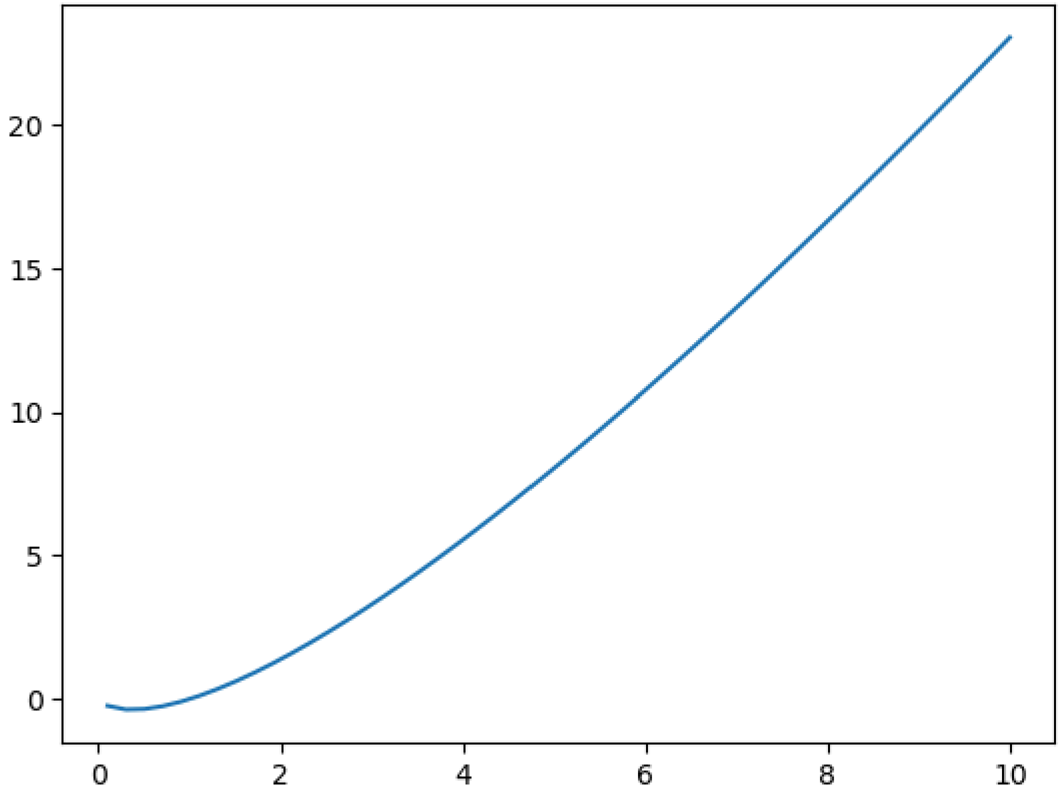
```
# y = sin(x)

x = np.linspace(-10,10,100)
y = np.sin(x)

plt.plot(x,y)
plt.show()
```




In [344…

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

y = x * np.log(x)

plt.plot(x,y)

plt.show()
```

```
C:\Users\sumit.DELL\AppData\Local\Temp\ipykernel_15772\944190992.py:3: RuntimeWar
ning: invalid value encountered in log
  y = x * np.log(x)
```

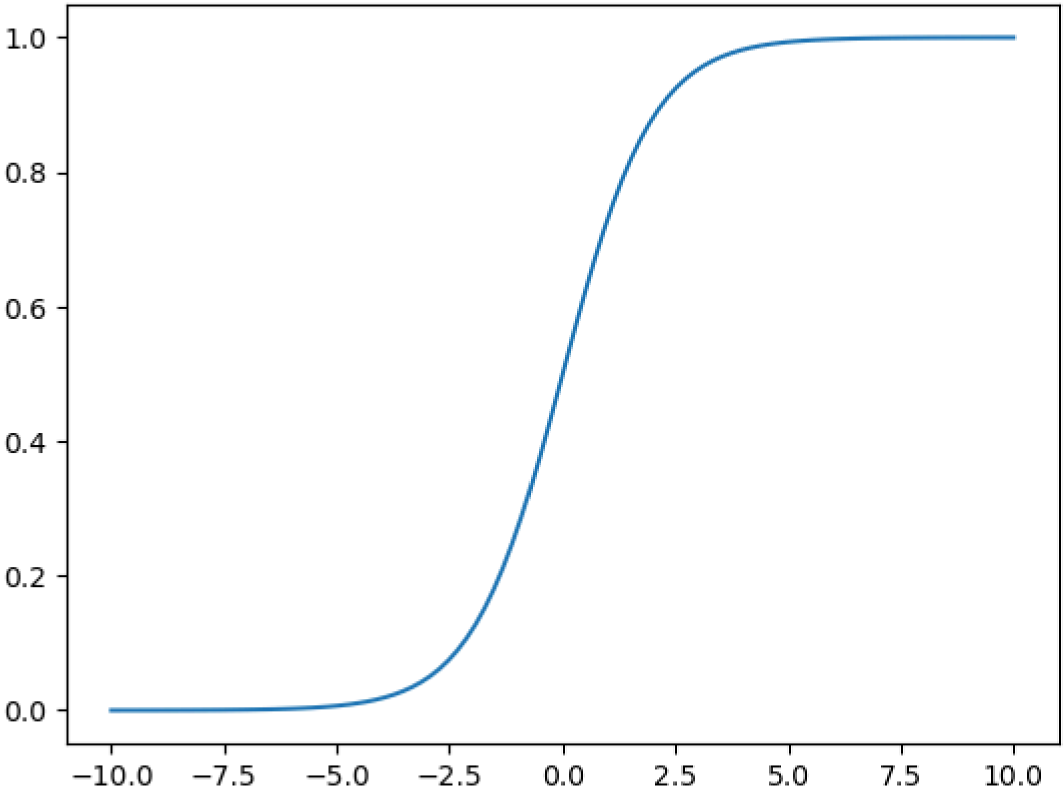


In [354…

```
# sigmoid

x = np.linspace(-10,10,100)
y = 1/(1+np.exp(-x))
plt.plot(x,y)
```

Out[354…

```
[<matplotlib.lines.Line2D at 0x26e058cbf50>]
```



In [356…

```
import numpy as np
```

In [358…
```
import matplotlib.pyplot as plt
```

## Meshgrid

Meshgrid are a way to **create coordinate matrices from coordinate vectors**. In numpy

- the meshgrid function is used to generate a coordinate grid given 1d coordinate arrays .it produces two 2d arrays representing the x and y coordinates of each point on the grid

the np.meshgrid function is used primarily for

- Creatig /plotting 2d functions f(x,y)
- Generating combinations of 2 or more numbers

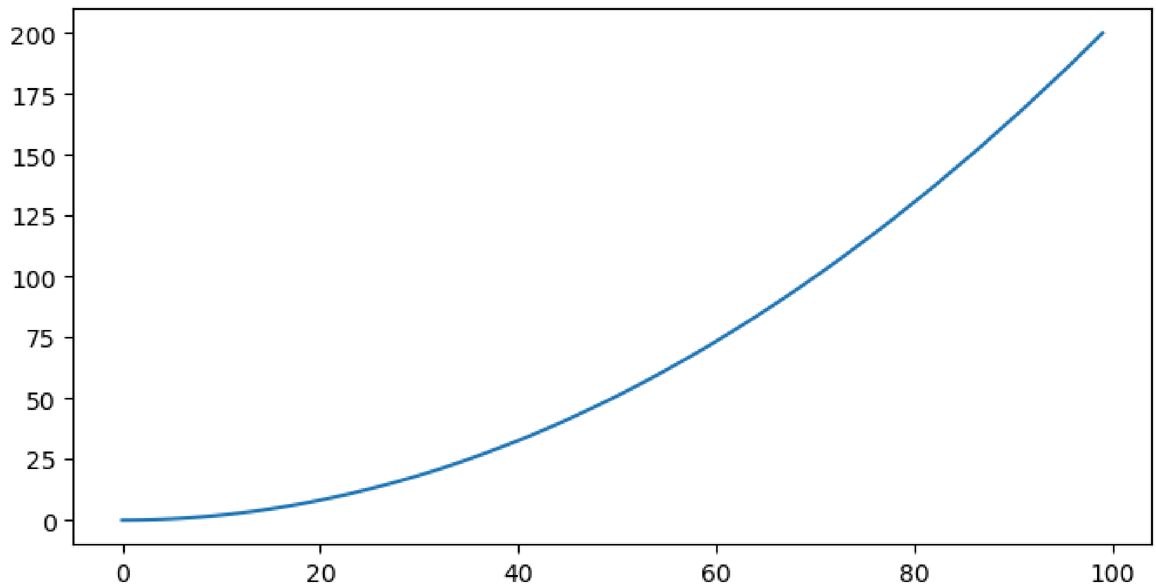
Example: How tou might think to create a 2d function f(x,y)

In [366…
```
x = np.linspace (0,10,100)
y = np.linspace(0,10,100)
```

Try to crate a 2d function

In [369…
```
f = x**2 + y**2
```

In [371…
```
plt.figure(figsize = (8,4))
plt.plot(f)
plt.show()
```




but f is a 1 dimansional function ! how does one generate a surface plot?

In [374…
```
x = np.arange (3)
```

In [376…
```
y = np.arange(3)

print(x)
print(y)
```

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

### Generating a meshgrid:

In [379…
```
xv , yv = np.meshgrid(x,y)
```

In [381…
```
xv
```

Out[381…
```
array([[0, 1, 2],
       [0, 1, 2],
       [0, 1, 2]])
```

In [383…
```
yv
```

Out[383…
```
array([[0, 0, 0],
       [1, 1, 1],
       [2, 2, 2]])
```

In [385…
```
P = np.linspace(-4,4,9)
V = np.linspace(-5,5,11)

print(P)
print(V)
```

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

In [387…
```
P_1 , V_1 = np.meshgrid(P,V)
```

In [389…
```
print(P_1)
```

```
[[-4. -3. -2. -1.  0.  1.  2.  3.  4.]
 [-4. -3. -2. -1.  0.  1.  2.  3.  4.]
 [-4. -3. -2. -1.  0.  1.  2.  3.  4.]
 [-4. -3. -2. -1.  0.  1.  2.  3.  4.]
 [-4. -3. -2. -1.  0.  1.  2.  3.  4.]
 [-4. -3. -2. -1.  0.  1.  2.  3.  4.]
 [-4. -3. -2. -1.  0.  1.  2.  3.  4.]
 [-4. -3. -2. -1.  0.  1.  2.  3.  4.]
 [-4. -3. -2. -1.  0.  1.  2.  3.  4.]
 [-4. -3. -2. -1.  0.  1.  2.  3.  4.]
 [-4. -3. -2. -1.  0.  1.  2.  3.  4.]]
```

In [391…
```
print(V_1)
```

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

## Numpy Meshgrid Creates Coordinates for a Grid system

these array , xv and yv each separately give the x and y coordinates on a 2d grid .you can do normal numpy operatios on these arrays :

In [394…
```
xv**2 +yv **2
```

Out[394…
```
array([[0, 1, 4],
       [1, 2, 5],
       [4, 5, 8]])
```

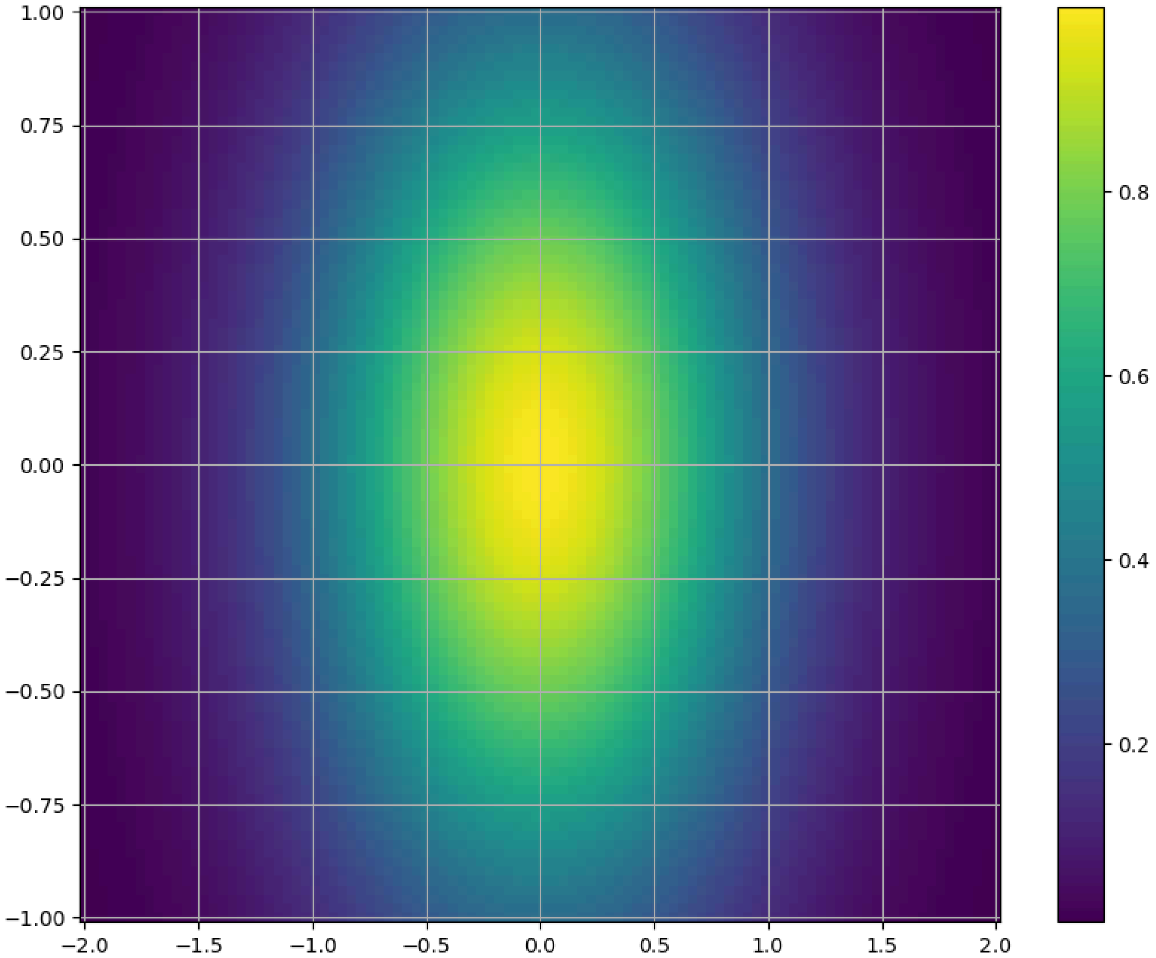
this can be done on alarger scale to plot surface plots of 2d functions

Generate functions f(x, y) = $e-(x^2+y^2)$ for $-2 \leq x \leq 2$ and $-1 \leq y \leq 1$

In [397…
```
x = np.linspace(-2,2,100)
y = np.linspace(-1,1,100)
xv,yv = np.meshgrid(x,y)

f = np.exp(-xv**2-yv**2)
```

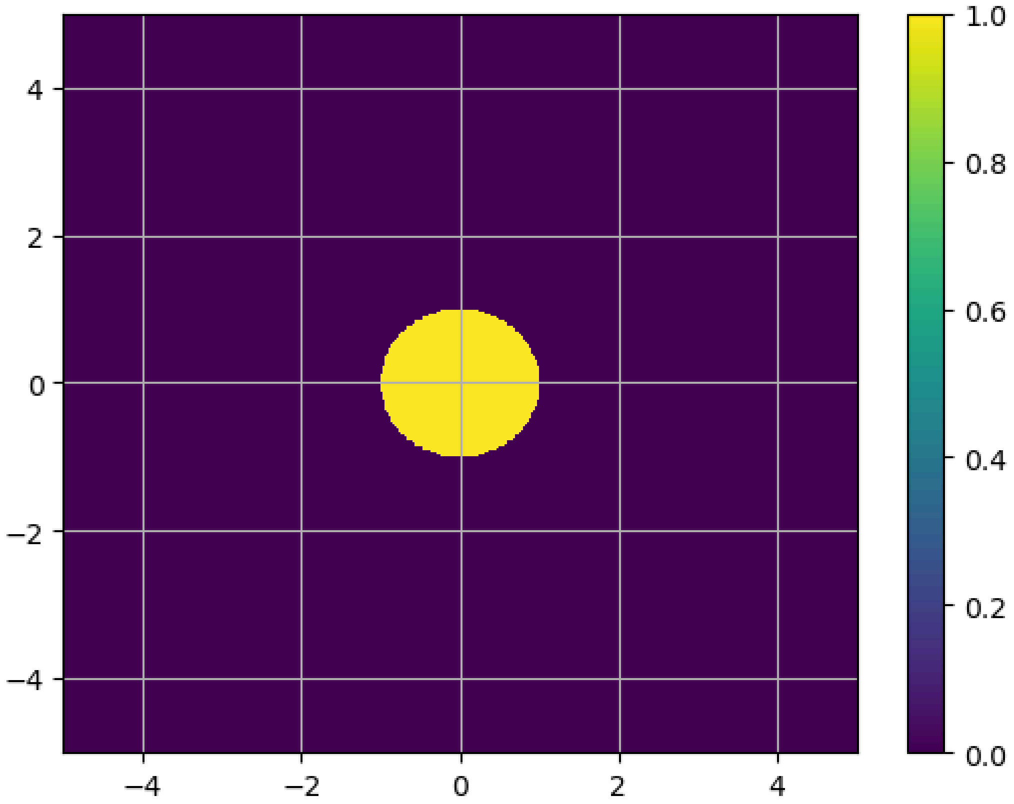
Note: pcolormesh is typically the preferable function 2d plotting , as opposed to imshow or pcolor, which take longer)

In [406…
```
plt.figure(figsize=(10,8))

plt.pcolormesh(xv,yv,f,shading = "nearest")

plt.colorbar()

plt.grid()
plt.show()
```




In [408…

```
import numpy as np

import matplotlib.pyplot as plt

def f (x,y):
    return np.where((x**2 +y**2 < 1 ) , 1.0 ,0.0)

x = np.linspace(-5,5,500)
y = np.linspace(-5,5,500)

xv,yv = np.meshgrid(x,y)

rectangular_mask = f(xv,yv)

plt.pcolormesh(xv,yv,rectangular_mask , shading = "auto")

plt.colorbar()
plt.grid()
plt.show()
```




In [410…

```
# numpy .linspace creates an array of 9 linearly placed elements between -4 and

x = np.linspace(-4,4,9)
```

In [412…

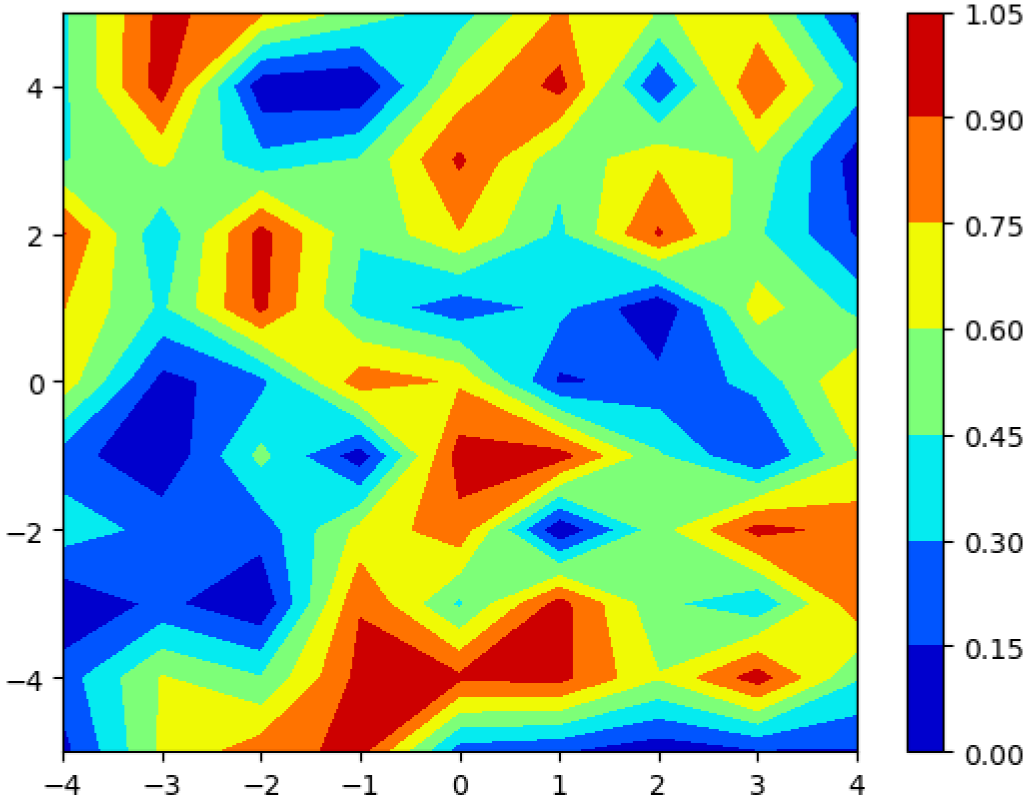
```
y = np.linspace(-5,5,11)
```

In [414…

```
x_1 ,y_1 = np.meshgrid(x,y)
```

In [418…

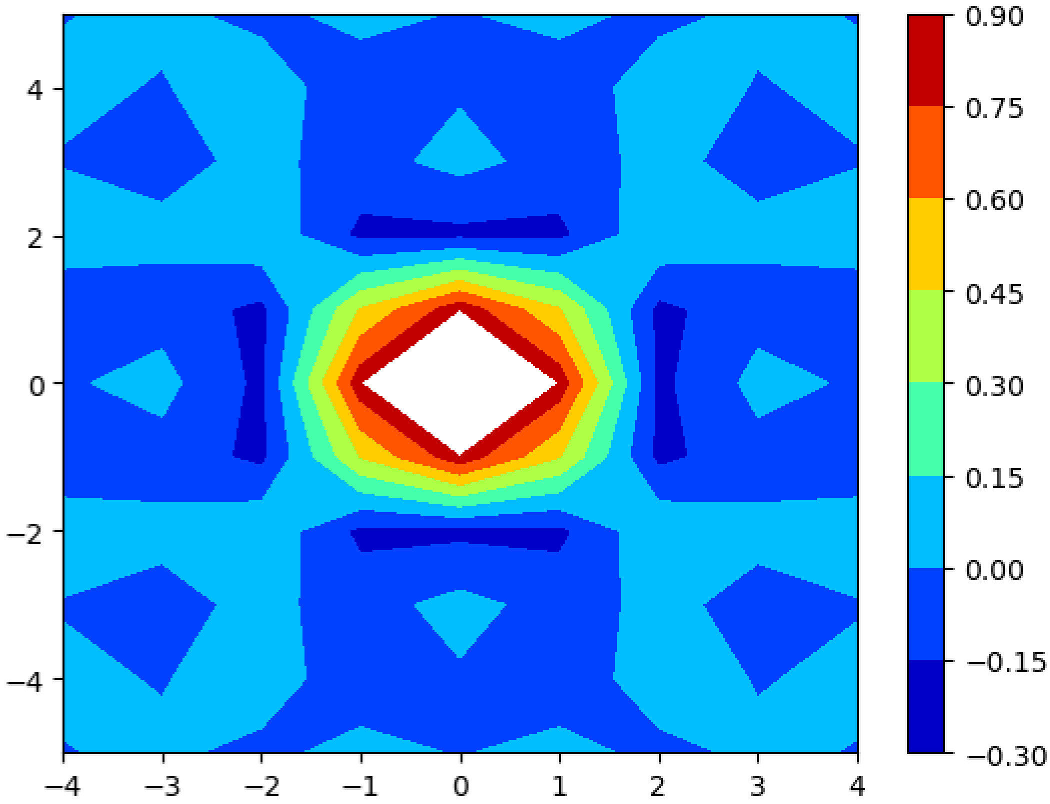
```
random_data = np.random.random((11,9))
plt.contourf(x_1 , y_1,random_data ,cmap = "jet")

plt.colorbar()

plt.show()
```




In [420…

```
sine  = (np.sin(x_1**2 + y_1**2))/ (x_1**2 + y_1**2)

plt.contourf(x_1,y_1,sine ,cmap = "jet")

plt.colororbar()

plt.show()
```

```
C:\Users\sumit.DELL\AppData\Local\Temp\ipykernel_15772\2718632641.py:1: RuntimeWa
rning: invalid value encountered in divide
  sine  = (np.sin(x_1**2 + y_1**2))/ (x_1**2 + y_1**2)
```




We observe that x_1 is a row repeated matrix whereas y_1 is a column repeated matrix .one row of x_1 and one column of y_1 is enough to determine the positions of all the points as the other values will get repeated over and over.

In [423…
```
x_1 ,y_1 = np.meshgrid(x,y,sparse = True)
```

In [425…
```
x_1
```

Out[425…
```
array([[-4., -3., -2., -1.,  0.,  1.,  2.,  3.,  4.]])
```

In [427…
```
y_1
```

Out[427…
```
array([[-5.],
       [-4.],
       [-3.],
       [-2.],
       [-1.],
       [ 0.],
       [ 1.],
       [ 2.],
       [ 3.],
       [ 4.],
       [ 5.]])
```

The shape of x_1 changed from (11,9) to (1,9) and that of y_1 changed from (11,9) to (11,1) the indexing of matrix is however different . Actually it is the exact opposite of cartesian indexing.

## np.sort

return a sorted copy of an array.

In [431…
```python
a = np.random.randint(1,100,15)
a
```
Out[431…
```
array([24, 83, 99, 35, 16, 75, 26, 31, 28, 79, 42, 34, 85, 66, 50])
```
In [433…
```python
b = np.random.randint(1,100,24).reshape(6,4)
b
```
Out[433…
```
array([[72, 38, 97,  3],
       [ 3, 62, 81, 81],
       [24, 27, 92, 31],
       [ 2, 78, 97, 74],
       [29, 99, 16, 31],
       [82,  8, 79, 45]])
```
In [435…
```python
np.sort(a) # default its ascending
```
Out[435…
```
array([16, 24, 26, 28, 31, 34, 35, 42, 50, 66, 75, 79, 83, 85, 99])
```
In [453…
```python
np.sort(a)[::-1]
```
Out[453…
```
array([99, 85, 83, 79, 75, 66, 50, 42, 35, 34, 31, 28, 26, 24, 16])
```
In [455…
```python
np.sort(b) # row wise sorting
```
Out[455…
```
array([[ 3, 38, 72, 97],
       [ 3, 62, 81, 81],
       [24, 27, 31, 92],
       [ 2, 74, 78, 97],
       [16, 29, 31, 99],
       [ 8, 45, 79, 82]])
```
In [457…
```python
np.sort(b,axis = 0)
```
Out[457…
```
array([[ 2,  8, 16,  3],
       [ 3, 27, 79, 31],
       [24, 38, 81, 31],
       [29, 62, 92, 45],
       [72, 78, 97, 74],
       [82, 99, 97, 81]])
```

## np.append

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

In [460…
```python
a
```
Out[460…
```
array([24, 83, 99, 35, 16, 75, 26, 31, 28, 79, 42, 34, 85, 66, 50])
```
In [462…
```python
np.append(a,200)
```
Out[462…
```
array([ 24,  83,  99,  35,  16,  75,  26,  31,  28,  79,  42,  34,  85,
        66,  50, 200])
```
In [464…
```python
b
```

Out[464...
```
array([[72, 38, 97,  3],
       [ 3, 62, 81, 81],
       [24, 27, 92, 31],
       [ 2, 78, 97, 74],
       [29, 99, 16, 31],
       [82,  8, 79, 45]])
```

In [466...
```
np.append(b,np.ones((b.shape[0],1)))
```

Out[466...
```
array([72., 38., 97.,  3.,  3., 62., 81., 81., 24., 27., 92., 31.,  2.,
       78., 97., 74., 29., 99., 16., 31., 82.,  8., 79., 45.,  1.,  1.,
        1.,  1.,  1.,  1.])
```

In [468...
```
np.append(b,np.ones((b.shape[0],1)),axis = 1)
```

Out[468...
```
array([[72., 38., 97.,  3.,  1.],
       [ 3., 62., 81., 81.,  1.],
       [24., 27., 92., 31.,  1.],
       [ 2., 78., 97., 74.,  1.],
       [29., 99., 16., 31.,  1.],
       [82.,  8., 79., 45.,  1.]])
```

In [472...
```
# adding random numbers in new columns

np.append(b,np.random.random((b.shape[0],1)),axis = 1)
```

Out[472...
```
array([[72.        , 38.        , 97.        ,  3.        ,  0.48045465],
       [ 3.        , 62.        , 81.        , 81.        ,  0.52846137],
       [24.        , 27.        , 92.        , 31.        ,  0.79535047],
       [ 2.        , 78.        , 97.        , 74.        ,  0.34142961],
       [29.        , 99.        , 16.        , 31.        ,  0.58971223],
       [82.        ,  8.        , 79.        , 45.        ,  0.48857017]])
```

## np.concatenate

numpy.concatenate() function concatenate a sequence of arrays along an existing axis.

In [475...
```
# code

c = np.arange(6).reshape(2,3)
d = np.arange(6,12).reshape(2,3)
```

In [477...
```
c
```

Out[477...
```
array([[0, 1, 2],
       [3, 4, 5]])
```

In [479...
```
d
```

Out[479...
```
array([[ 6,  7,  8],
       [ 9, 10, 11]])
```

we can use it replacement of vstack and hstaxk

In [482...
```
np.concatenate((c,d))
```

Out[482…
```
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11]])
```

In [488…
```
np.concatenate((c,d ), axis = 1)
```

Out[488…
```
array([[ 0,  1,  2,  6,  7,  8],
       [ 3,  4,  5,  9, 10, 11]])
```

In [490…
```
np.concatenate((c,d),axis =1)
```

Out[490…
```
array([[ 0,  1,  2,  6,  7,  8],
       [ 3,  4,  5,  9, 10, 11]])
```

## np.unique

with the help of np.unique () method , we can get the unique values froma n array given as parameter in np.unique() method.

In [494…
```
e = np.array([1,1,2,2,3,3,4,4,5,5,6,6,])
e
```

Out[494…
```
array([1, 1, 2, 2, 3, 3, 4, 4, 5, 5, 6, 6])
```

In [496…
```
np.unique(e)
```

Out[496…
```
array([1, 2, 3, 4, 5, 6])
```

## np.expand_dims

In [499…
```
a
```

Out[499…
```
array([24, 83, 99, 35, 16, 75, 26, 31, 28, 79, 42, 34, 85, 66, 50])
```

In [501…
```
a.shape
```

Out[501…
```
(15,)
```

In [505…
```
np.expand_dims(a,axis = 0).shape
```

Out[505…
```
(1, 15)
```

In [507…
```
np.expand_dims(a,axis = 1)
```

Out[507…
```
array([[24],
       [83],
       [99],
       [35],
       [16],
       [75],
       [26],
       [31],
       [28],
       [79],
       [42],
       [34],
       [85],
       [66],
       [50]])
```

we can use in row vector and column vector .

expand_dims() is used to insert an addition dimension in input tensor

In [510…
```
np.expand_dims(a,axis = 1 ) .shape
```

Out[510…
```
(15, 1)
```

### np.where

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

In [513…
```
a
```

Out[513…
```
array([24, 83, 99, 35, 16, 75, 26, 31, 28, 79, 42, 34, 85, 66, 50])
```

In [515…
```
np.where(a>50)
```

Out[515…
```
(array([ 1,  2,  5,  9, 12, 13], dtype=int64),)
```

In [521…
```
np.where(a>50,0,a)# replace all values > 50 with zero
```

Out[521…
```
array([24,  0,  0, 35, 16,  0, 26, 31, 28,  0, 42, 34,  0,  0, 50])
```

In [523…
```
# print and replace all even numbers to 0

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

Out[523…
```
array([ 0, 83, 99, 35,  0, 75,  0, 31,  0, 79,  0,  0, 85,  0,  0])
```

### np.argmax

the numpy.argmax() function returns indices of the max element of the array in a particular axis

arg = argument

In [526…
```
a
```

Out[526…
```
array([24, 83, 99, 35, 16, 75, 26, 31, 28, 79, 42, 34, 85, 66, 50])
```

In [530…
```
np.argmax(a) # biggest number : index number
```

Out[530…
```
2
```

In [532…
```
b # on 2d
```

Out[532…
```
array([[72, 38, 97,  3],
       [ 3, 62, 81, 81],
       [24, 27, 92, 31],
       [ 2, 78, 97, 74],
       [29, 99, 16, 31],
       [82,  8, 79, 45]])
```

In [534…
```
np.argmax(b,axis =1 ) # row wise biggest number : index
```

Out[534…
```
array([2, 2, 2, 2, 1, 0], dtype=int64)
```

In [538…
```
np.argmax(b,axis = 0) #"column wise"
```

Out[538…
```
array([5, 4, 0, 1], dtype=int64)
```

In [540…
```
a
```

Out[540…
```
array([24, 83, 99, 35, 16, 75, 26, 31, 28, 79, 42, 34, 85, 66, 50])
```

In [542…
```
np.argmin(a)
```

Out[542…
```
4
```

## On Statistics :

### np.cumsum

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

In [545…
```
a
```

Out[545…
```
array([24, 83, 99, 35, 16, 75, 26, 31, 28, 79, 42, 34, 85, 66, 50])
```

In [547…
```
 np.cumsum(a)
```

Out[547…
```
array([ 24, 107, 206, 241, 257, 332, 358, 389, 417, 496, 538, 572, 657,
       723, 773])
```

In [549…
```
b
```

Out[549…
```
array([[72, 38, 97,  3],
       [ 3, 62, 81, 81],
       [24, 27, 92, 31],
       [ 2, 78, 97, 74],
       [29, 99, 16, 31],
       [82,  8, 79, 45]])
```

In [551…
```
np.cumsum(b,axis = 1) # row wise calculation or cumulative sum
```

Out[551…
```
array([[ 72, 110, 207, 210],
       [  3,  65, 146, 227],
       [ 24,  51, 143, 174],
       [  2,  80, 177, 251],
       [ 29, 128, 144, 175],
       [ 82,  90, 169, 214]])
```

In [553…
```
np.cumsum(b,axis = 0) # column wise calculation or cumulative sum
```

Out[553…
```
array([[ 72,  38,  97,   3],
       [ 75, 100, 178,  84],
       [ 99, 127, 270, 115],
       [101, 205, 367, 189],
       [130, 304, 383, 220],
       [212, 312, 462, 265]])
```

In [555…
```
# np.cumprod ---> multiply

a
```

Out[555…
```
array([24, 83, 99, 35, 16, 75, 26, 31, 28, 79, 42, 34, 85, 66, 50])
```

In [557…
```
np.cumprod(a)
```

Out[557…
```
array([         24,        1992,      197208,     6902280,   110436480,
        -307198592,   602771200,  1506038016,  -780608512, -1538530304,
        -193763328,  2001981440, -1630269440,  -223600640,  1704869888])
```

In [559…
```
np.percentile(a,100) # max
```

Out[559…
```
99.0
```

In [561…
```
np.percentile(a,0)
```

Out[561…
```
16.0
```

## np.percentile

numpy.percentile() function useed to compute the nth percentile ot hte given data (array elements ) along the specified axis.

In [564…
```
np.percentile(a,50)
```

Out[564…
```
42.0
```

In [566…
```
a
```

Out[566…
```
array([24, 83, 99, 35, 16, 75, 26, 31, 28, 79, 42, 34, 85, 66, 50])
```

In [570…
```
(a*50) / 100
```

Out[570…
```
array([12. , 41.5, 49.5, 17.5,  8. , 37.5, 13. , 15.5, 14. , 39.5, 21. ,
       17. , 42.5, 33. , 25. ])
```

In [578…
```
k = (a*50) / 100
```

In [580…
```
k
```

Out[580…
```
array([12. , 41.5, 49.5, 17.5,  8. , 37.5, 13. , 15.5, 14. , 39.5, 21. ,
       17. , 42.5, 33. , 25. ])
```

In [586…
```
np.median(a)
```

Out[586…
```
42.0
```

## np.histogram

Numpy has a built in numpy.histogram() function which represents the **frequency of data** distribution in the graphical form

In [589…
```
a
```

Out[589…
```
array([24, 83, 99, 35, 16, 75, 26, 31, 28, 79, 42, 34, 85, 66, 50])
```

In [591…
```
np.histogram(a,bins = [10,20,30,40,50,60,70,80,90,100])
```

Out[591…
```
(array([1, 3, 3, 1, 1, 1, 2, 2, 1], dtype=int64),
 array([ 10,  20,  30,  40,  50,  60,  70,  80,  90, 100]))
```

In [593…
```
np.histogram(a,bins = [0,50,100])
```

Out[593…
```
(array([8, 7], dtype=int64), array([  0,  50, 100]))
```

## np.corrcoef

return pearson product moment correlation coefficients.

In [600…
```
salary = np.array([20000,40000,25000,35000,60000])
experience = np.array([1,3,2,4,2])
```

In [602…
```
salary
```

Out[602…
```
array([20000, 40000, 25000, 35000, 60000])
```

In [604…
```
experience
```

Out[604…
```
array([1, 3, 2, 4, 2])
```

In [606…
```
np.corrcoef(salary,experience) # correlation coeficient
```

Out[606…
```
array([[1.        , 0.25344572],
       [0.25344572, 1.        ]])
```

## Utility functions

### np.isin

with the help of numpy.isin method we can see that one array having values are checked in a different numpy array having different elements with different sizes.

In [609…
```
a
```

Out[609…
```
array([24, 83, 99, 35, 16, 75, 26, 31, 28, 79, 42, 34, 85, 66, 50])
```

In [611…
```
items = [10,20,30,40,50,60,70,80,90,100]

np.isin(a,items)
```

Out[611…
```
array([False, False, False, False, False, False, False, False, False,
       False, False, False, False, False,  True])
```

In [613…
```
a[np.isin(a,items)]
```

Out[613…
```
array([50])
```

## np.flip

the numpy.flip() function reverses the order of array elements along the specified axis preserving the shape of the array.

In [616…
```
a
```

Out[616…
```
array([24, 83, 99, 35, 16, 75, 26, 31, 28, 79, 42, 34, 85, 66, 50])
```

In [618…
```
np.flip(a)
```

Out[618…
```
array([50, 66, 85, 34, 42, 79, 28, 31, 26, 75, 16, 35, 99, 83, 24])
```

In [620…
```
b
```

Out[620…
```
array([[72, 38, 97,  3],
       [ 3, 62, 81, 81],
       [24, 27, 92, 31],
       [ 2, 78, 97, 74],
       [29, 99, 16, 31],
       [82,  8, 79, 45]])
```

In [622…
```
np.flip(b)
```

Out[622…
```
array([[45, 79,  8, 82],
       [31, 16, 99, 29],
       [74, 97, 78,  2],
       [31, 92, 27, 24],
       [81, 81, 62,  3],
       [ 3, 97, 38, 72]])
```

In [624…
```
np.flip(b,axis =1)
```

Out[624…
```
array([[ 3, 97, 38, 72],
       [81, 81, 62,  3],
       [31, 92, 27, 24],
       [74, 97, 78,  2],
       [31, 16, 99, 29],
       [45, 79,  8, 82]])
```

In [626…
```
np.flip(b,axis = 0)
```

Out[626…
```
array([[82,  8, 79, 45],
       [29, 99, 16, 31],
       [ 2, 78, 97, 74],
       [24, 27, 92, 31],
       [ 3, 62, 81, 81],
       [72, 38, 97,  3]])
```

## np.put

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

In [630…
```
a
```

Out[630…
```
array([24, 83, 99, 35, 16, 75, 26, 31, 28, 79, 42, 34, 85, 66, 50])
```

In [634…
```
np.put(a,[0,1],[110,530]) # permanent changes
```

In [637…
```
a
```

Out[637…
```
array([110, 530,  99,  35,  16,  75,  26,  31,  28,  79,  42,  34,  85,
        66,  50])
```

## np.delete

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

In [640…
```
a
```

Out[640…
```
array([110, 530,  99,  35,  16,  75,  26,  31,  28,  79,  42,  34,  85,
        66,  50])
```

In [642…
```
np.delete(a,0)# deleted 0 index item
```

Out[642…
```
array([530,  99,  35,  16,  75,  26,  31,  28,  79,  42,  34,  85,  66,
        50])
```

In [644…
```
a
```

Out[644…
```
array([110, 530,  99,  35,  16,  75,  26,  31,  28,  79,  42,  34,  85,
        66,  50])
```

In [648…
```
np.delete(a,[0,2,4] )# deleted 0,2,4 index items
```

Out[648…
```
array([530,  35,  75,  26,  31,  28,  79,  42,  34,  85,  66,  50])
```

## set functions

- np.union1d
- np.intersect1d
- np.setdiff1d
- np.setxor1d
- np.in1d

In [651…
```
m = np.array([1,2,3,4,5])
n = np.array([3,4,5,6,7])
```

In [653…
```
np.union1d(m,n)
```

Out[653…
```
array([1, 2, 3, 4, 5, 6, 7])
```

In [655…
```
np.intersect1d(m,n)
```

Out[655…
```
array([3, 4, 5])
```

In [657…
```
np.setdiff1d(m,n)
```

Out[657…
```
array([1, 2])
```

In [659…
```
np.setdiff1d(n,m)
```

Out[659…
```
array([6, 7])
```

In [661…
```
np.setxor1d(m,n)
```

Out[661…
```
array([1, 2, 6, 7])
```

In [663…
```
m[np.in1d(m,1)]
```

Out[663…
```
array([1])
```

In [665…
```
np.in1d(m,10)
```

Out[665…
```
array([False, False, False, False, False])
```

## np.clip

numpy.clip() function is used to clip limit the values in an array.

In [668…
```
a
```

Out[668…
```
array([110, 530,  99,  35,  16,  75,  26,  31,  28,  79,  42,  34,  85,
        66,  50])
```

In [670…
```
np.clip(a,a_min = 15,a_max =50)
```

Out[670…
```
array([50, 50, 50, 35, 16, 50, 26, 31, 28, 50, 42, 34, 50, 50, 50])
```

**it clips the minimum data to 15 and replaces everything below data to 15 and maximum to 50**

## np.swapaxes

numpy.swapaxes() function interchange two axes of an arrray.

In [677…
```
arr = np.array([[1,2,3],[4,5,6]])
swapped_arr = np.swapaxes(arr,0,1)
```

In [679…
```
arr
```

Out[679…
```
array([[1, 2, 3],
       [4, 5, 6]])
```

In [681…
```
swapped_arr
```

Out[681…
```
array([[1, 4],
       [2, 5],
       [3, 6]])
```

In [683…
```
print("Original array:")
print(arr)
```

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

In [685…
```
print("swapped array:")
print(swapped_arr)
```

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

---

In [ ]: