Numpy

Creating numpy array

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

In [2]:
```
np.array ([2,4,56,422,32,1]) # 1d array
```

Out[2]:
```
array([  2,   4,  56, 422,  32,   1])
```

In [3]:
```
a = np.array([2,4,56,422,32,1]) # vector
print(a)
```

```
[  2   4  56 422  32   1]
```

In [4]:
```
type(a)
```

Out[4]:
```
numpy.ndarray
```

In [5]:
```
#  2D array (matrix)

new = np.array([[ 45,34,22,2],[24,55,3,22]])
print(new)
```

```
[[45 34 22  2]
 [24 55  3 22]]
```

In [6]:
```
# 3D ----- # Tensor

np.array ([[2,22,33,4,45],[23,45,56,66,2],[357,523,32,24,2],[32,32,44,33,234]])
```

Out[6]:
```
array([[  2,  22,  33,   4,  45],
       [ 23,  45,  56,  66,   2],
       [357, 523,  32,  24,   2],
       [ 32,  32,  44,  33, 234]])
```

dtype

The desired data-type for the array. if not given,then the type will be determined as the minimum type requried to hold the objects in the sequences.

In [7]:
```
np.array ([11,23,44],dtype=float)
```

Out[7]:
```
array([11., 23., 44.])
```

In [8]:
```
np.array ([11,23,44],dtype=bool) # Here true becoz , python treats Non - Zero
```

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

In [9]:
```
np.array([11,23,44],dtype=complex)
```

Out[9]:
```
array([11.+0.j, 23.+0.j, 44.+0.j])
```

Numpy arrays vs python sequence

arange

arange can be called with a varying number of positional arguments

In [10]:
```
np.arange(1,25) # 1-included ,25 -last one got excluded
```

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

In [11]:
```
np.arange(1,25,2) # strides---> Alternate number
```

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

reshape

Both of number products should be umber of items present inside the array

In [12]:
```
np.arange(1,11).reshape (5,2) # converted 5 rows 2 coulms
```

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

In [13]:
```
np.arange(1,11).reshape (2,5) # converted 2 rows 5 coulms
```

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

In [14]:
```
np.arange(1,13).reshape (3,4) # converted 3 rows 4 coulms
```

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

ones & zeros

you can initialize the values and create values ex:in deep learning weight shape

In [15]:
```
# np.ones and np. zeros

np.ones((3,4)) # we have mention inside tuple
```

```
Out[15]: array([[1., 1., 1., 1.],
                [1., 1., 1., 1.],
                [1., 1., 1., 1.]])
```

```
In [16]: np.zeros((3,4))
```

```
Out[16]: array([[0., 0., 0., 0.],
                [0., 0., 0., 0.],
                [0., 0., 0., 0.]])
```

```
In [17]: # Another type ---> random()
         np.random.random((4,3))
```

```
Out[17]: array([[0.68623804, 0.48277622, 0.48002681],
                [0.81338799, 0.79790827, 0.04529834],
                [0.74861688, 0.79676704, 0.77240913],
                [0.71710111, 0.40305542, 0.45143362]])
```

linspace

It is also called as linearly space ,linearly separable ,in a given range at equal distance it creates points

```
In [18]: np.linspace (-10,10,10) # here:lower range, upper range number of items to gen
```

```
Out[18]: array([-10.        ,  -7.77777778,  -5.55555556,  -3.33333333,
                 -1.11111111,   1.11111111,   3.33333333,   5.55555556,
                  7.77777778,  10.        ])
```

```
In [19]: np.linspace(-2,12,6)
```

```
Out[19]: array([-2. ,  0.8,  3.6,  6.4,  9.2, 12. ])
```

identity

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

```
In [20]: # creating  the identity matrix
         np.identity (3)
```

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

```
In [21]: np.identity(6)
```

```
Out[21]: 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.]])
```

Array Attributes

In [22]:
```
a1 = np.arange(10) # 1D
a1
```

Out[22]:
```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

In [23]:
```
a2 = np.arange (12,dtype = float).reshape(3,4) # matrix
a2
```

Out[23]:
```
array([[ 0.,  1.,  2.,  3.],
       [ 4.,  5.,  6.,  7.],
       [ 8.,  9., 10., 11.]])
```

In [24]:
```
a3 = np.arange (8).reshape (2,2,2) # 3D ---> Tensor
a3
```

Out[24]:
```
array([[[0, 1],
        [2, 3]],

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

ndim

To findout given arrays number of dimensions

In [25]:
```
a1.ndim
```

Out[25]:
```
1
```

In [26]:
```
a2.ndim
```

Out[26]:
```
2
```

In [27]:
```
a3.ndim
```

Out[27]:
```
3
```

shape

gives each item consist of. no.of rows and np.of column

In [28]:
```
a1.shape # 1D array has 10 items
```

Out[28]:
```
(10,)
```

In [29]:
```
a2.shape # 3 rows and 4 colums
```

Out[29]:
```
(3, 4)
```

In [30]:
```
a3.shape # first,2 says it consints of 2D arrays .2,2 gives no .of rows and c
```

Out[30]:
```
(2, 2, 2)
```

size

gives number of items

In [31]:
```
a3
```

Out[31]:
```
array([[[0, 1],
        [2, 3]],

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

In [32]:
```
a3.size # it has 8 items . like shape : 2,2,2, = 8
```

Out[32]:
```
8
```

In [33]:
```
a2
```

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

In [34]:
```
a2.size
```

Out[34]:
```
12
```

item size

memory occupied by the item

In [35]:
```
a1
```

Out[35]:
```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

In [36]:
```
a1.itemsize # bytes
```

Out[36]:
```
8
```

In [37]:
```
a2.itemsize # integer 64 gives = 8 bytes
```

Out[37]:
```
8
```

In [38]:
```
a3.itemsize # integer 32 gives = 4 bytes
```

Out[38]:
```
8
```

dtype

gives data type of the item

In [39]:
```
print(a1.dtype)
print(a2.dtype)
print(a3.dtype)
```

```
int64
float64
int64
```

Changeing Data Type

In [40]:
```
#astype
x = np.array([33,22,2.5])
x
```

Out[40]:
```
array([33. , 22. ,  2.5])
```

In [41]:
```
x.astype(int)
```

Out[41]:
```
array([33, 22,  2])
```

Array Operations

In [42]:
```
z1 = np.arange(12).reshape(3,4)
z2 = np.arange(12,24).reshape(3,4)
```

In [43]:
```
z1
```

Out[43]:
```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

In [44]:
```
z2
```

Out[44]:
```
array([[12, 13, 14, 15],
       [16, 17, 18, 19],
       [20, 21, 22, 23]])
```

Scalar Operations

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

In [45]:
```
# arithmetic
z1 + 2
```

Out[45]:
```
array([[ 2,  3,  4,  5],
       [ 6,  7,  8,  9],
       [10, 11, 12, 13]])
```

In [46]:
```
# subtracion
z1 - 2
```

Out[46]:
```
array([[-2, -1,  0,  1],
       [ 2,  3,  4,  5],
       [ 6,  7,  8,  9]])
```

In [47]:
```
# multiplication
z1 * 2
```

Out[47]:
```
array([[ 0,  2,  4,  6],
       [ 8, 10, 12, 14],
       [16, 18, 20, 22]])
```

In [48]:
```
# power
z1 ** 2
```

Out[48]:
```
array([[  0,   1,   4,   9],
       [ 16,  25,  36,  49],
       [ 64,  81, 100, 121]])
```

In [49]:
```
## modulo
z1 % 2
```

Out[49]:
```
array([[0, 1, 0, 1],
       [0, 1, 0, 1],
       [0, 1, 0, 1]])
```

Relational Operators

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 [50]:
```
z2
```

Out[50]:
```
array([[12, 13, 14, 15],
       [16, 17, 18, 19],
       [20, 21, 22, 23]])
```

In [51]:
```
z2 > 2 # if 2 is greater than everything gives true
```

Out[51]:
```
array([[ True,  True,  True,  True],
       [ True,  True,  True,  True],
       [ True,  True,  True,  True]])
```

In [52]:
```
z2 > 20
```

Out[52]:
```
array([[False, False, False, False],
       [False, False, False, False],
       [False,  True,  True,  True]])
```

vector operators

we can apply on both numpy array

In [53]:
```
z1
```

Out[53]:
```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

In [54]:
```
z2
```

Out[54]:
```
array([[12, 13, 14, 15],
       [16, 17, 18, 19],
       [20, 21, 22, 23]])
```

In [55]:
```
# Arithemetic
z1 + z2 # both numpy array shape is same , we can add item wise
```

Out[55]:
```
array([[12, 14, 16, 18],
       [20, 22, 24, 26],
       [28, 30, 32, 34]])
```

In [56]:
```
z1 * z2
```

Out[56]:
```
array([[  0,  13,  28,  45],
       [ 64,  85, 108, 133],
       [160, 189, 220, 253]])
```

In [57]:
```
z1 - z2
```

Out[57]:
```
array([[-12, -12, -12, -12],
       [-12, -12, -12, -12],
       [-12, -12, -12, -12]])
```

In [58]:
```
z1 / z2
```

Out[58]:
```
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 Functions

In [59]:
```
k1 = np.random.random((3,3))
k1 = np.round(k1*100)
k1
```

Out[59]:
```
array([[36., 20., 89.],
       [26., 95., 33.],
       [14., 63., 20.]])
```

In [60]:
```
# Max
np.max(k1)
```

Out[60]:
```
np.float64(95.0)
```

In [61]:
```
# min
np.min(k1)
```

Out[61]:
```
np.float64(14.0)
```

In [62]:
```
# sum
np.sum(k1)
```

Out[62]: np.float64(396.0)

In [63]:
```
# prod ----> multiplication
np.prod(k1)
```

Out[63]: np.float64(92136556512000.0)

In Numpy

0 = column , 1 = row

In [64]:
```
# if we want maximum of every row
np.max(k1, axis = 1)
```

Out[64]: array([89., 95., 63.])

In [65]:
```
# maximum of every colum
np.max(k1,axis = 0)
```

Out[65]: array([36., 95., 89.])

In [66]:
```
# product of every column
np.prod(k1, axis=0)
```

Out[66]: array([13104., 119700., 58740.])

Statistics related functions

In [67]:
```
# mean
k1
```

Out[67]:
```
array([[36., 20., 89.],
       [26., 95., 33.],
       [14., 63., 20.]])
```

In [68]:
```
np.mean(k1)
```

Out[68]: np.float64(44.0)

In [69]:
```
# mean of every column
k1.mean(axis=0)
```

Out[69]: array([25.33333333, 59.33333333, 47.33333333])

In [70]:
```
# median
np.median(k1)
```

Out[70]: np.float64(33.0)

In [71]:
```
np.median(k1,axis = 1)
```

Out[71]:
```
array([36., 33., 20.])
```

In [72]:
```
# standard deviation
np.std(k1)
```

Out[72]:
```
np.float64(28.959742171964628)
```

In [73]:
```
np.std(k1,axis=0)
```

Out[73]:
```
array([ 8.99382504, 30.72819914, 29.93697083])
```

In [74]:
```
# variance
np.var(k1)
```

Out[74]:
```
np.float64(838.6666666666666)
```

Trignometry Functions

In [75]:
```
np.sin(k1) # sin
```

Out[75]:
```
array([[-0.99177885,  0.91294525,  0.86006941],
       [ 0.76255845,  0.68326171,  0.99991186],
       [ 0.99060736,  0.1673557 ,  0.91294525]])
```

In [76]:
```
np.cos(k1)
```

Out[76]:
```
array([[-0.12796369,  0.40808206,  0.51017704],
       [ 0.64691932,  0.73017356, -0.01327675],
       [ 0.13673722,  0.98589658,  0.40808206]])
```

In [77]:
```
np.tan(k1)
```

Out[77]:
```
array([[  7.75047091,   2.23716094,   1.68582537],
       [  1.17875355,   0.93575247, -75.3130148 ],
       [  7.24460662,   0.16974975,   2.23716094]])
```

Dot Product

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

In [78]:
```
s2 = np.arange(12).reshape(3,4)
s3 = np.arange(12,24).reshape(4,3)
```

In [79]:
```
s2
```

Out[79]:
```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

In [80]:
```
s3
```

Out[80]:
```
array([[12, 13, 14],
       [15, 16, 17],
       [18, 19, 20],
       [21, 22, 23]])
```

In [81]:
```
np.dot(s2,s3) # dot product of s2,s3
```

Out[81]:
```
array([[114, 120, 126],
       [378, 400, 422],
       [642, 680, 718]])
```

Log and Exponents

In [82]:
```
np.exp(s2)
```

Out[82]:
```
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 / floor / cell

1.round

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

In [83]:
```
# Round to the nearest integer
arr = np.array([1.2, 2.7, 3.5, 4.9])
rounded_arr = np.round(arr)
print(rounded_arr)
```

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

In [84]:
```
# 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 [85]:
```
# randomly
np.round(np.random.random((2,3))*100)
```

Out[85]:
```
array([[29., 35., 23.],
       [89., 13., 42.]])
```

2.Floor

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

In [86]:
```
# Floor operation
arr = np.array ([1.2, 2.7, 3.5, 4.9])
floored_arr = np.floor(arr)
print(floored_arr)
```

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

In [87]:
```
np.floor(np.random.random((2,3))*100) # gives the smallest integer ex :6.8=
```

Out[87]:
```
array([[62., 47., 43.],
       [93., 36., 24.]])
```

3.Ceil

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

In [88]:
```
arr = np.array([1.2,2.7,3.5,4.9])
ceiled_arr = np.ceil(arr)
print(ceiled_arr)
```

```
[2. 3. 4. 5.]
```

In [89]:
```
np.ceil(np.random.random((2,3))*100) # gives highest integer ex:7.8=8
```

Out[89]:
```
array([[27., 58., 63.],
       [40., 98., 58.]])
```

4.Indexing and slicing

In [90]:
```
p1 = np.arange(10)
p2 = np.arange(12).reshape(3,4)
p3 = np.arange(8).reshape(2,2,2)
```

In [91]:
```
p1
```

Out[91]:
```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

In [92]:
```
p2
```

Out[92]:
```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

In [93]:
```
p3
```

Out[93]:
```
array([[[0, 1],
        [2, 3]],

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

Indexing on 1D array

In [94]:
```
p1
```

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

In [95]:
```
# Fetching last item
p1[-1]
```

Out[95]:
```
np.int64(9)
```

In [96]:
```
# Fetching first item
p1[0]
```

Out[96]:
```
np.int64(0)
```

Indexing on 2D array

In [97]:
```
p2
```

Out[97]:
```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

In [98]:
```
# fetching desire element : 6
p2[1,2] # here 1 = row(second), 2= column(third), becoz it starts from zero
```

Out[98]:
```
np.int64(6)
```

In [99]:
```
# fetching desire element : 11
p2[2,3] # row=2 , column=3
```

Out[99]:
```
np.int64(11)
```

In [100…
```
# fetching desired element :4
p2[1,0] # row =1, column =0
```

Out[100…
```
np.int64(4)
```

Indexing on 3D (Tensors)

In [101…
```
p3
```

Out[101…
```
array([[[0, 1],
        [2, 3]],

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

In [102…
```
# fetching desire element :5
p3[1,0,1]
```

Out[102…
```
np.int64(5)
```

EXPLANATION: Here 3D is consists of 2, 2D array , so Firstly we take 1 because our desired is 5 is in second matrix which is 1 .and 1 row so () and second column so 1

In [103…
```
# fetching desired element :2
p3[0,1,0]
```

Out[103…
```
np.int64(2)
```

EXPLANATION :Here firstly we take () because our desired is 2,is in first matrix which is (). and 2 row so 1 and first column so ()

In [104…
```
# fetching desired element :0
p3 [0,0,0]
```

Out[104…
```
np.int64(0)
```

Here first we take () because our desire is (),is in first matrix which is ().and 1 row so () and first column so ()

In [105…
```
# fetching desired element :6
p3[1,1,0]
```

Out[105…
```
np.int64(6)
```

EXPLANATION: Here first we take, because our desired is 6,is in second matrix which is 1. and second row so 1 and first column so ()

Slicing

fetching multiple items

Slicing on 1D

In [106…
```
p1
```

Out[106…
```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

In [107…
```
   # fetching desired elements are : 2,3,4
p1[2:5]
```

Out[107…
```
array([2, 3, 4])
```

EXPLANATION: Here first we take, whatever we need first item ,2 and up last(4) +1 which 5 .because last element is not included

In [108…
```
# Alternate (same as python)
p1[2:5:2]
```

Out[108…
```
array([2, 4])
```

slicing on 2D

In [109…
```
p2
```
Out[109…
```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```
In [110…
```
# fetching total first row
p2[0, :]
```
Out[110…
```
array([0, 1, 2, 3])
```
EXPLANATION:Here we want rows so (:),and we want 3rd column so 2

In [111…
```
# fetching total third column
p2[:,2]
```
Out[111…
```
array([ 2,  6, 10])
```
In [112…
```
# fetch 5,6 and 9,10
p2
```
Out[112…
```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```
In [113…
```
p2[1:3] # for rows
```
Out[113…
```
array([[ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```
In [114…
```
p2[1:3,1:3] # for columns
```
Out[114…
```
array([[ 5,  6],
       [ 9, 10]])
```
EXPLANATION:Here first [1:3] we slice 2 second row is to third row is not existed which is 2 and secondly, we take [1:3] which is same as first:we slice 2 second row is to third row is not included which is 3

In [115…
```
# fetch 0,3 and 8,11
p2
```
Out[115…
```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```
In [116…
```
p2[::2, ::3]
```
Out[116…
```
array([[ 0,  3],
       [ 8, 11]])
```
EXPLANATION:Here we take(:) because we want all rows, second(:2) for alternate value, and (:) for all columns and (:3) jump for two steps

In [117…
```
# fetch 1,3 and 9,11
p2
```
Out[117…
```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```
In [118…
```
p2[::2] # for rows
```
Out[118…
```
array([[ 0,  1,  2,  3],
       [ 8,  9, 10, 11]])
```
In [119…
```
p2[::2, 1::2] # columns
```
Out[119…
```
array([[ 1,  3],
       [ 9, 11]])
```
EXPLANATION:Here we take (:) because we want all rows , second(:2) for alternate value, and (1) for we want from second column and (:2) jump for two steps and ignore middle one

In [120…
```
# fetch only 4,7
p2
```
Out[120…
```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```
In [121…
```
p2[1] # first rows
```
Out[121…
```
array([4, 5, 6, 7])
```
In [122…
```
p2[1,::3] # second columns
```
Out[122…
```
array([4, 7])
```
EXPLANATION:Here we take (1) because we want second row , second (:) for total column, (:3) jump for two steps and ignore middle ones

In [123…
```
# fetch 1,2,3 and 5,6,7
p2
```
Out[123…
```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```
In [124…
```
p2[0:2] # first fetched rows
```
Out[124…
```
array([[0, 1, 2, 3],
       [4, 5, 6, 7]])
```
In [125…
```
p2[0:2, 1:] # for column
```
Out[125…
```
array([[1, 2, 3],
       [5, 6, 7]])
```
In [126…
```
# fetch 1,3 and 5,7
```

```
p2
```

Out[126…

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

In [127…

```
p2[0:2] # for rows
```

Out[127…

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

In [128…

```
p2[0:2, 1::2]
```

Out[128…

```
array([[1, 3],
       [5, 7]])
```

slicing in 3D

In [129…

```
p3 = np.arange(27).reshape(3,3,3)
p3
```

Out[129…

```
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 [130…

```
# fetch second matrix
p3[1]
```

Out[130…

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

In [131…

```
# fetch first and last
p3[::2]
```

Out[131…

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

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

EXPLANATION: Along the first axis, (::2) selects every second element. This means it will select the subarrays at indices 0 and 2

In [132…

```
# fetch 1 2d array 's 2 row ---> 3,4,5
p3
```

Out[132…
```
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 [133…
```
p3[0] # first numpy array
```

Out[133…
```
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
```

In [134…
```
p3[0,1,:]
```

Out[134…
```
array([3, 4, 5])
```

EXPLANATION : 0 represents first matrix, 1 represents second row ,(:) means total

In [135…
```
# Fetch 2 numpy array , middle column ---> 10,13,16
p3
```

Out[135…
```
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 [136…
```
p3[1] # middle array
```

Out[136…
```
array([[ 9, 10, 11],
       [12, 13, 14],
       [15, 16, 17]])
```

In [137…
```
p3[1,:,1]
```

Out[137…
```
array([10, 13, 16])
```

EXPLANATION: 1 respresnts middle column , (:) all columns, 1 represents middle column

In [138…
```
# Fetch 3 array----> 22,23,25,26
p3
```

Out[138…
```
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 [139…
```
p3[2] # last row
```

Out[139…
```
array([[18, 19, 20],
       [21, 22, 23],
       [24, 25, 26]])
```

In [140…
```
p3 [2,1:] # last two rows
```

Out[140…
```
array([[21, 22, 23],
       [24, 25, 26]])
```

In [141…
```
p3[2, 1:, 1:] # last two columns
```

Out[141…
```
array([[22, 23],
       [25, 26]])
```

EXPLANATION:Here we go through 3 stages, where 2 for last array , and (1:) from second row to total rows , and (1:)nis for second column to total columns

In [142…
```
# Fetch o,2,18,20
p3
```

Out[142…
```
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 [143…
```
p3[0::2] # for array
```

Out[143…
```
array([[[ 0,  1,  2],
        [ 3,  4,  5],
        [ 6,  7,  8]],

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

In [144…
```
p3[0::2 ,0] # for rows
```

Out[144…
```
array([[ 0,  1,  2],
       [18, 19, 20]])
```

In [145…
```
p3[0::2, 0,::2] # for columns
```

Out[145…
```
array([[ 0,  2],
       [18, 20]])
```

EXPLANATION: Here we take (0::2) first and last column , so we did jump using this , and we took (0) for first row , and we(::2) ignored middle column

Iterating

In [146…
```
p1
```

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

In [147…
```
# Looping on 1D array
for i in p1:
    print(i)
```

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

In [148…
```
p2
```

Out[148…
```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

In [149…
```
## Looping on 2D array

for i in p2:
    print(i) # Prints rows
```

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

In [150…
```
p3
```

Out[150…
```
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 [151…
```
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]]
```

print all items in 3D using nditer----> first convert in to 1D and applying loop

In [152…
```
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 to columns and columns into rows

In [153…
```
p2
```

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

In [154…
```
np.transpose(p2)
```

Out[154…
```
array([[ 0,  4,  8],
       [ 1,  5,  9],
       [ 2,  6, 10],
       [ 3,  7, 11]])
```

In [155…
```
# Another method
p2.T
```

Out[155…
```
array([[ 0,  4,  8],
       [ 1,  5,  9],
       [ 2,  6, 10],
       [ 3,  7, 11]])
```

In [156…
```
p3
```

Out[156…
```
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 [157…
```
p3.T
```

Out[157…
```
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 1D

In [158…
```
p2
```

Out[158…
```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

In [159…
```
p2.ravel()
```

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

In [160…
```
p3
```

Out[160…
```
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 [161…
```
p3.ravel()
```

Out[161…
```
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])
```

Stacking

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

In [162…
```
# Horizantal stacking

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

In [163…
```
w1
```

Out[163…
```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

In [164…
```
w2
```

Out[164…
```
array([[12, 13, 14, 15],
       [16, 17, 18, 19],
       [20, 21, 22, 23]])
```

using hstack for horizantal stacking

In [165…
```
np.hstack((w1,w2))
```

Out[165…
```
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 [166…
```
# Vertical stacking
w1
```

Out[166…
```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

In [167…
```
w2
```

Out[167…
```
array([[12, 13, 14, 15],
       [16, 17, 18, 19],
       [20, 21, 22, 23]])
```

using Vstack for vertical stacking

In [168…
```
np.vstack ((w1,w2))
```

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]])
```

Splitting

its opposite of stacking

In [169…
```
# Horizantal splitting
w1
```

Out[169…
```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

In [170…
```
np.hsplit(w1,2)
```

Out[170…
```
[array([[0, 1],
        [4, 5],
        [8, 9]]),
 array([[ 2,  3],
        [ 6,  7],
        [10, 11]])]
```

In [171…
```
np.hsplit(w1,4) # splitting by 4
```

Out[171…
```
[array([[0],
        [4],
        [8]]),
 array([[1],
        [5],
        [9]]),
 array([[ 2],
        [ 6],
        [10]]),
 array([[ 3],
        [ 7],
        [11]])]
```

In [172…
```
# vertical splitting
w2
```

Out[172…
```
array([[12, 13, 14, 15],
       [16, 17, 18, 19],
       [20, 21, 22, 23]])
```

In [173…
```
np.vsplit(w2,3) # splitting into 3 rows
```

Out[173…
```
[array([[12, 13, 14, 15]]),
 array([[16, 17, 18, 19]]),
 array([[20, 21, 22, 23]])]
```

Numpy Arrays vs python sequences

Speed of List vs Numpy

list

In [174…
```
# Element-wise addition

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)
```

```
4.863830804824829
```

Numpy

In [175…
```
import numpy as np

a = np.arange(10000000)
b = np.arange(10000000,20000000)

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

```
0.36379528045654297
```

In [176…
```
2.7065064907073975 / 0.02248692512512207
```

Out[176…
```
120.35911871666826
```

Memory used for list vs Numpy

List

In [177…
```
p = [i for i in range(10000000)]

import sys

sys.getsizeof(p)
```

Out[177…
```
89095160
```

Numpy

In [178…
```
R = np.arange(10000000)

sys.getsizeof(R)
```

Out[178…
```
80000112
```

In [179…
```
# we can decrease more in numpy

R = np.arange(10000000, dtype=np.int16)

sys.getsizeof(R)
```

Out[179…
```
20000112
```

In [180…
```
# Normal Indexing and slicing

w = np.arange(12).reshape(4,3)
w
```

Out[180…
```
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11]])
```

In [181...
```python
# fetching 5 from array
w[1,2]
```
Out[181...
```
np.int64(5)
```
In [182...
```python
# fetching 4,5,,7,8
w[1:3]
```
Out[182...
```
array([[3, 4, 5],
       [6, 7, 8]])
```
In [183...
```python
w[1:3, 1:3]
```
Out[183...
```
array([[4, 5],
       [7, 8]])
```

Fancy Indexing

In [184...
```python
w
```
Out[184...
```
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11]])
```
In [185...
```python
# fetch 1,3,4 row
w[[0,2,3]]
```
Out[185...
```
array([[ 0,  1,  2],
       [ 6,  7,  8],
       [ 9, 10, 11]])
```
In [186...
```python
# New array

z = np.arange(24).reshape(6,4)
z
```
Out[186...
```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15],
       [16, 17, 18, 19],
       [20, 21, 22, 23]])
```
In [187...
```python
# fetch 1, 3, 4, 6 rows

z[[0,2,3,5]]
```
Out[187...
```
array([[ 0,  1,  2,  3],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15],
       [20, 21, 22, 23]])
```
In [188...
```python
# fetch 1,3,4 columns

z[:,[0,2,3]]
```

Out[188…

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

Boolean Indexing

In [189…

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

In [190…

```
G
```

Out[190…

```
array([[36, 84,  2, 43],
       [24, 33, 77, 35],
       [29, 45, 88, 15],
       [47, 34, 25,  3],
       [10, 32, 77, 83],
       [40, 78, 49, 64]], dtype=int32)
```

In [191…

```
# find all numbers greater than 50
G > 50
```

Out[191…

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

In [192…

```
# Where is True, it gives result, everything other than removed.we got value
G[G > 50]
```

Out[192…

```
array([84, 77, 88, 77, 83, 78, 64], dtype=int32)
```

it is best Techinque to filter than data in given condition

In [193…

```
# find out even numbers
G % 2 == 0
```

Out[193…

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

In [194…

```
# find all numbers greater than 50 and are even
(G > 50) & (G % 2 == 0)
```

Out[194…

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

Here we used(&)bitwise not logical(and), because we are working with boolean values

In [195…
```
# Result
G[(G > 50) & (G % 2 == 0)]
```

Out[195…
```
array([84, 88, 78, 64], dtype=int32)
```

In [196…
```
# find all numbers not divisible by 7
G % 7 == 0
```

Out[196…
```
array([[False,  True, False, False],
       [False, False,  True,  True],
       [False, False, False, False],
       [False, False, False, False],
       [False, False,  True, False],
       [False, False,  True, False]])
```

In [197…
```
# Result
G[~(G % 7 == 0)] # (~) = Not
```

Out[197…
```
array([36,  2, 43, 24, 33, 29, 45, 88, 15, 47, 34, 25,  3, 10, 32, 83, 40,
       78, 64], dtype=int32)
```

Broadcasting

In [198…
```
# same shape
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 [199…
```
# diff shape
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]]
```

Broadcasting Rules

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

. if the numbers of dimensions of the two arrays are different, add new dimensions with size 1 to the head of the array with the smaller dimension.

ex: (3,4)=2D, (3)=1D--->convert into (1,3) (3,3,3)=3D, (3)=1D---> convert into (1,1,3)

2.Make each dimension of the two arrays do not match, dimensions with size 1 are stretched to the size of the other array. ex:(3,3)=2D,(3)=1D---> CONVERTED (1,3) than strech to (3,3)

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

In [200…

```
# More examples

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

print(a) # 2D
```

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

In [201…

```
print(b) # 1D
```

```
[0 1 2]
```

In [202…

```
print(a+b) # Arthematic operation
```

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

EXPLANATION:Arthematic 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 [203…

```
# Could not Broadcast

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

print(a)
print(b)

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

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
[0 1 2]
[[ 0  2  4  6]
 [ 8 10 12 14]
 [16 18 20 22]]
```

EXPLANATION:Arthematic operation not 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 equals to b. so it got failed

In [204…

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

print(a)
print(b)

print(a+b)
```

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

EXPLANATION:Arthematic operation possible because, Here a = (1,3) is 2D and b = (3,1) is 2D so did converted (1,3) to (3,3) and b(3,1) converted(1) to 3 than (3,3).finally it equally.

In [205…

```
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]]
```

EXPLANATION:Same as before

In [206…

```
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 [207…

```
import numpy as np

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

print(a + b.T)
```

```
[[ 0  4  8 12]
 [ 5  9 13 17]
 [10 14 18 22]]
```

EXPLANATION:there is no 1 to convert, so got failed

In [208…

```
import numpy as np

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

b2 = np.tile(b, (2,2))   # repeat b to make 4×4

print(a + b2)
```

```
[[ 0  2  2  4]
 [ 6  8  8 10]
 [ 8 10 10 12]
 [14 16 16 18]]
```

EXPLANATION:there is no 1 to convert, so got failed

Working with mathematical formulas

In [209…

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

In [210…

```
k
```

Out[210…

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

In [211…

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

Out[211…

```
np.int64(45)
```

In [212…

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

Out[212…

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

Sigmoid

In [213...
```
def sigmoid(array):
    return 1/(1+np.exp(-(array)))
k = np.arange(10)
sigmoid(k)
```

Out[213...
```
array([0.5       , 0.73105858, 0.88079708, 0.95257413, 0.98201379,
       0.99330715, 0.99752738, 0.99908895, 0.99966465, 0.99987661])
```

In [214...
```
k = np.arange(100)
sigmoid(k)
```

Out[214...
```
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 [215...
```
actual = np.random.randint(1,50,25)
predicted = np.random.randint(1,50,25)
actual
```

Out[215...
```
array([21, 40, 47, 22, 45,  7, 19, 17,  2, 11, 42, 19, 39, 27, 21, 35, 11,
       27, 47, 19, 46, 31, 33,  5, 17], dtype=int32)
```

In [216...
```
predicted
```

Out[216...
```
array([23, 27, 10, 28, 21,  1,  8, 35, 21, 14, 15,  3, 25, 44,  1,  3, 16,
       42, 19, 31,  8, 40, 14, 42, 33], dtype=int32)
```

In [217...
```
def mse(actual,predicted):
    return np.mean((actual-predicted)**2)

mse(actual,predicted)
```

Out[217...
```
np.float64(423.52)
```

In [218…
```
# detailed
actual-predicted
```

Out[218…
```
array([ -2,  13,  37,  -6,  24,   6,  11, -18, -19,  -3,  27,  16,  14,
       -17,  20,  32,  -5, -15,  28, -12,  38,  -9,  19, -37, -16],
      dtype=int32)
```

In [219…
```
(actual-predicted)**2
```

Out[219…
```
array([   4,  169, 1369,   36,  576,   36,  121,  324,  361,    9,  729,
        256,  196,  289,  400, 1024,   25,  225,  784,  144, 1444,   81,
        361, 1369,  256], dtype=int32)
```

In [220…
```
np.mean((actual-predicted)**2)
```

Out[220…
```
np.float64(423.52)
```

Working with missing values

In [221…
```
# working with missing values->np.nan

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

Out[221…
```
array([ 1.,  2.,  3.,  4., nan,  6.])
```

In [222…
```
np.isnan(s)
```

Out[222…
```
array([False, False, False, False,  True, False])
```

In [223…
```
s[np.isnan(s)] # Nan values
```

Out[223…
```
array([nan])
```

In [224…
```
s[~np.isnan(s)] # Not Non Values
```

Out[224…
```
array([1., 2., 3., 4., 6.])
```

Plotting Graphs

In [225…
```
# plotting a 2D pot
# x = y

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

Out[225…
```
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.        ])
```

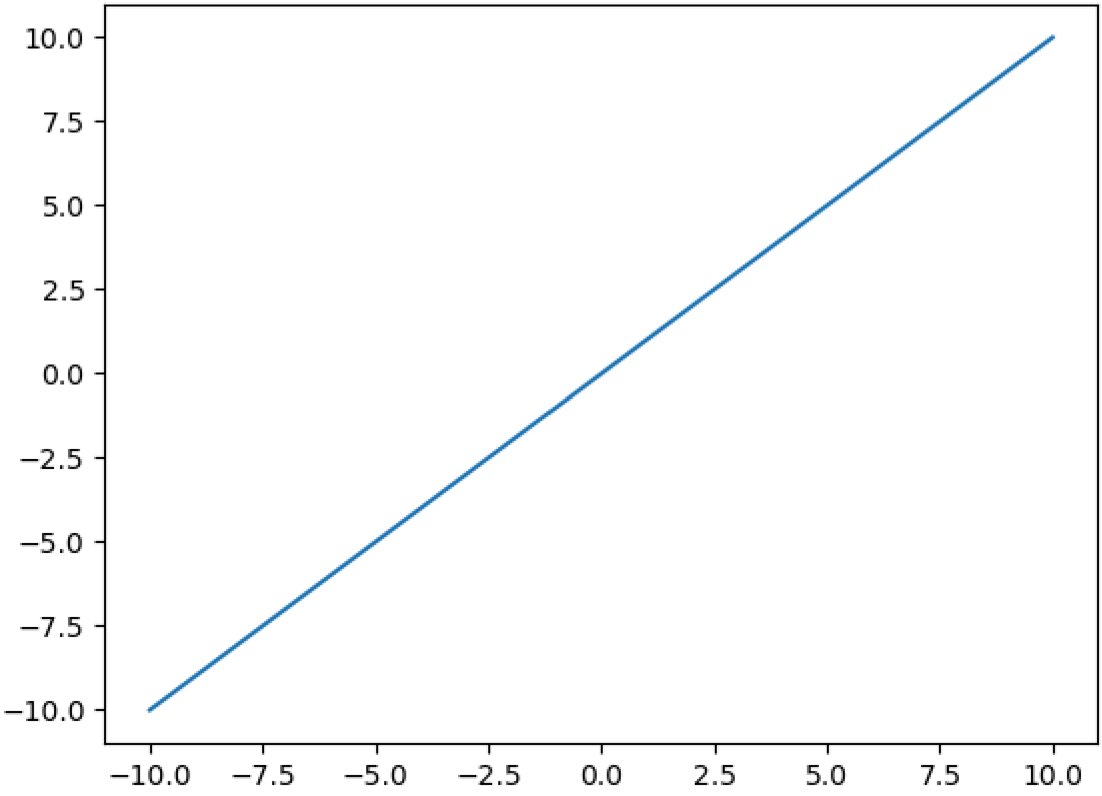
In [226…
```
y = x
```

In [227…
```
y
```

Out[227…
```
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 [228…
```
import matplotlib.pyplot as plt
plt.plot(x,y)
```

Out[228… [<matplotlib.lines.Line2D at 0x29048c17890>]



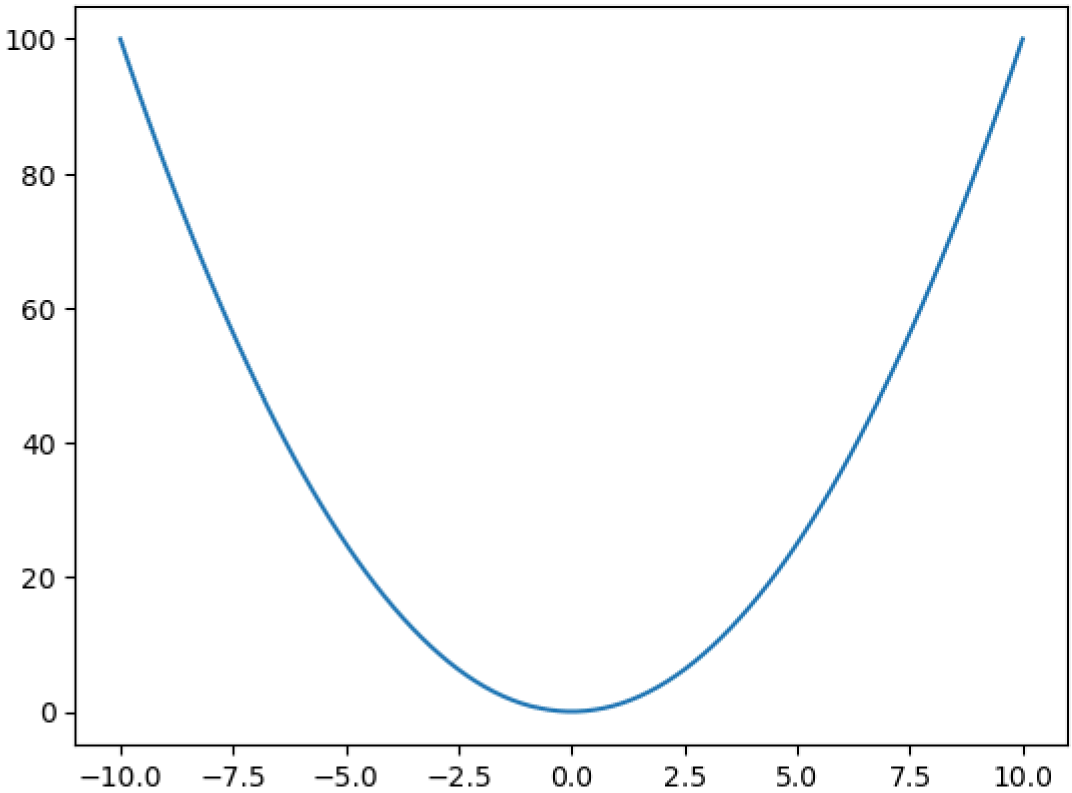
In [229…

```
# y = x^2

x = np.linspace(-10,10,100)
y = x**2

plt.plot(x,y)
```

Out[229… [<matplotlib.lines.Line2D at 0x2904a54a490>]

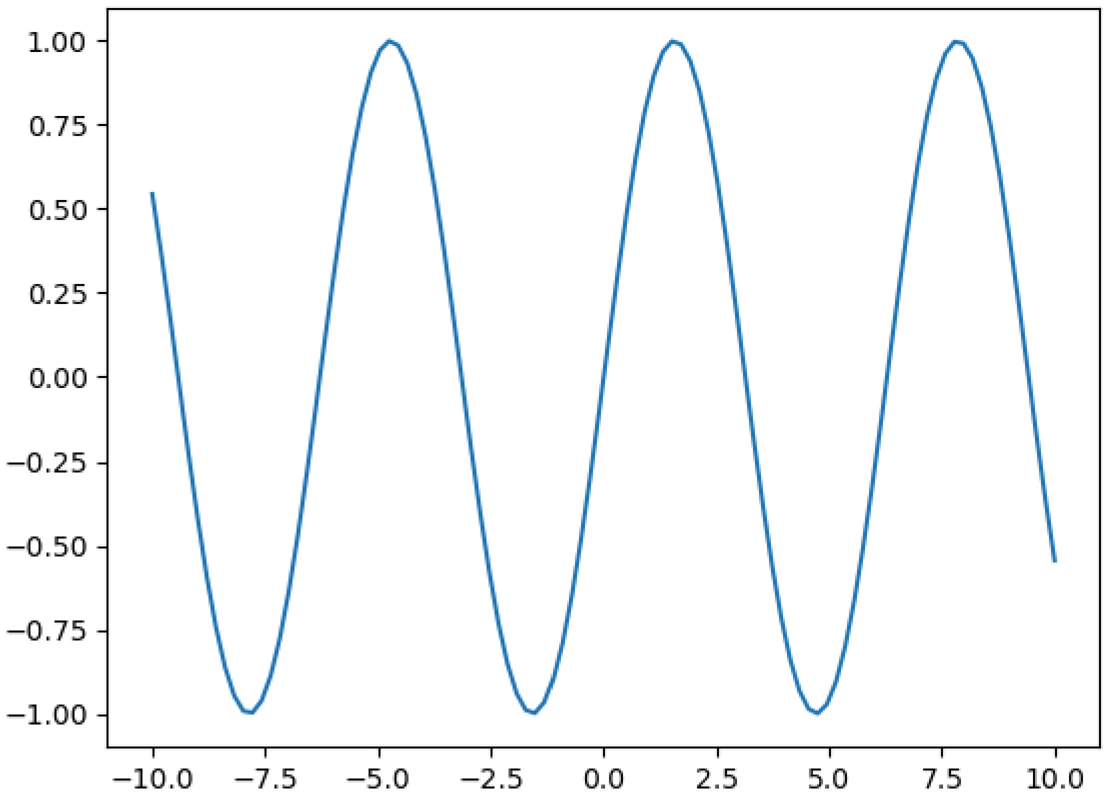


In [230…
```
# y = sin(x)

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

plt.plot(x,y)
```

Out[230…
```
[<matplotlib.lines.Line2D at 0x2904a5d2350>]
```

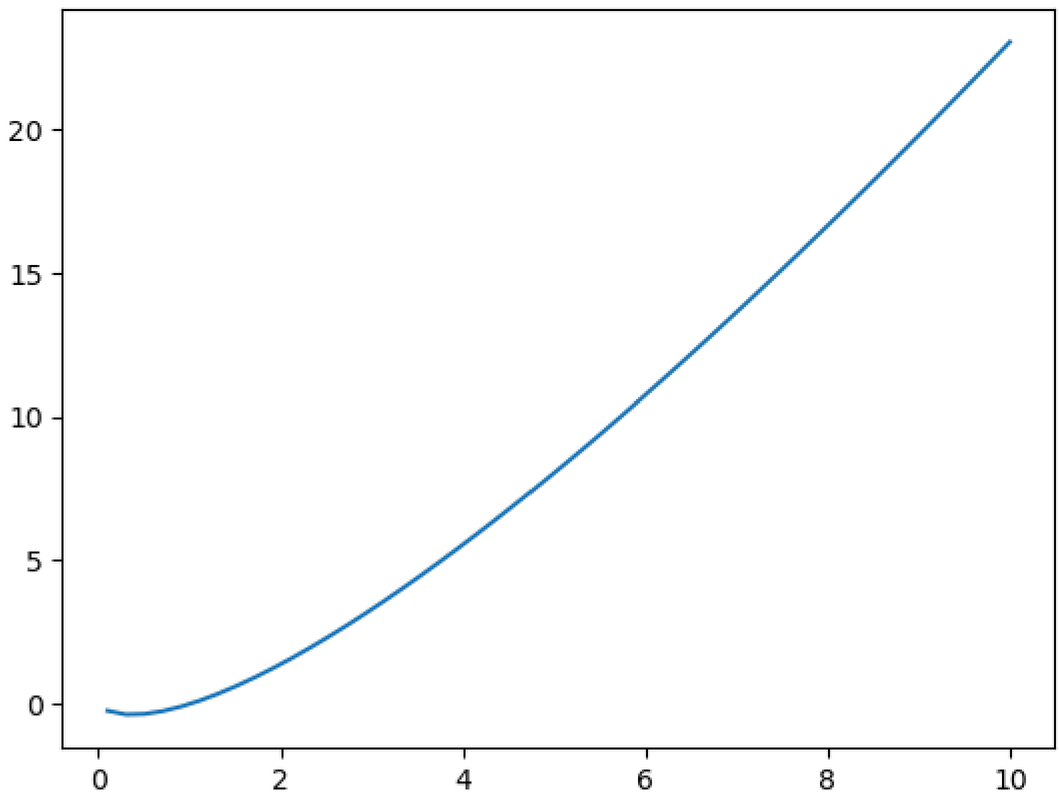


In [231…
```
# y = xlog(x)
x = np.linspace(-10,10,100)
y = x*np.log(x)

plt.plot(x,y)
```

```
C:\Users\Lenovo\AppData\Local\Temp\ipykernel_18184\3903925937.py:3: RuntimeWarnin
g: invalid value encountered in log
  y = x*np.log(x)
```

Out[231…
```
[<matplotlib.lines.Line2D at 0x2904a65ae90>]
```



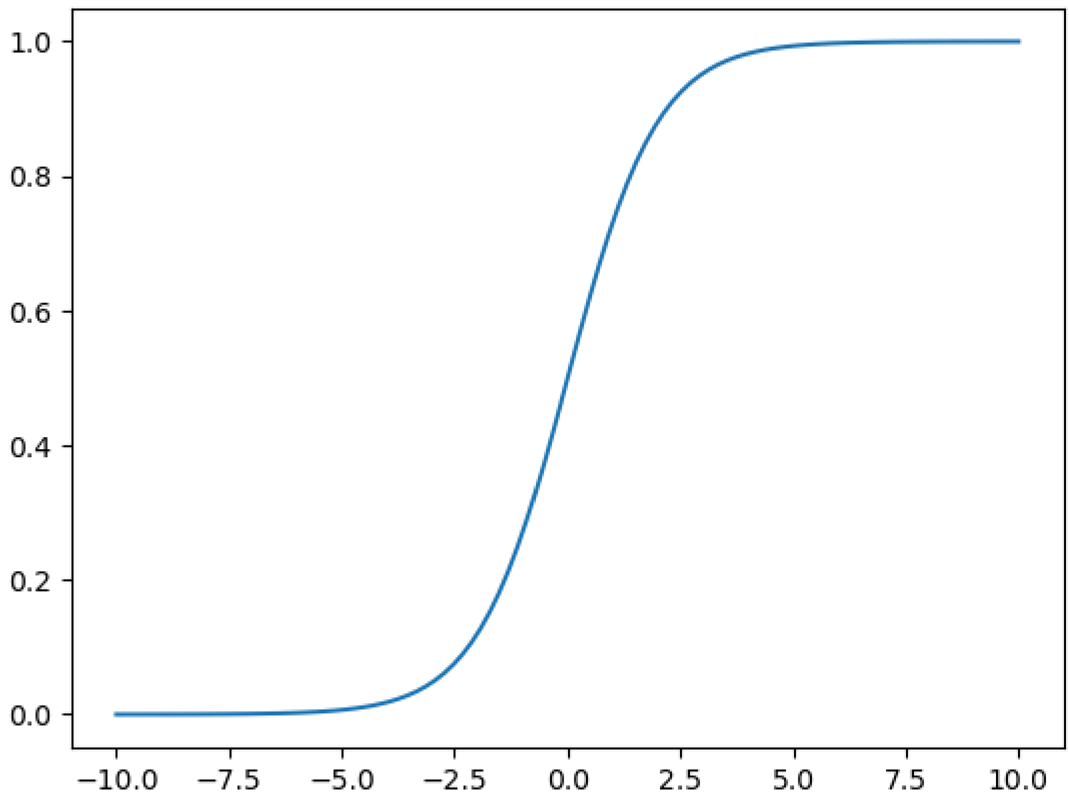
In [232…

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

plt.plot(x,y)
```

Out[232…

```
[<matplotlib.lines.Line2D at 0x2904a6e4550>]
```



In [233…

```
import numpy as np
```

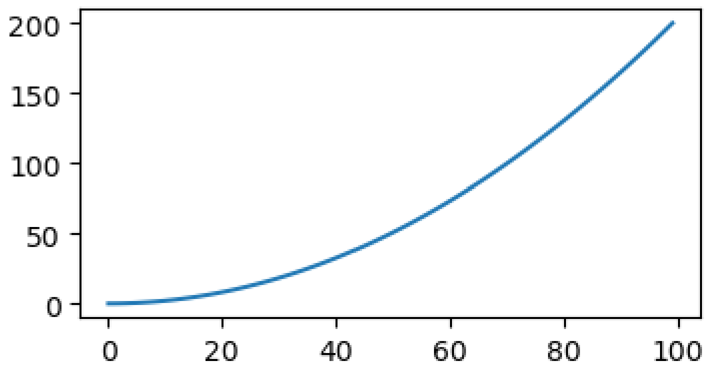
```
import matplotlib.pyplot as plt
```

Meshgrid

In [234…
```
x = np.linspace(0,10,100)
y = np.linspace(0,10,100)
```

In [235…
```
f = x**2+y**2
```

In [236…
```
plt.figure(figsize=(4,2))
plt.plot(f)
plt.show()
```



But f is a 1 dimensional function! How does one generate a surface plot?

In [237…
```
x = np.arange(3)
y = np.arange(3)
```

In [238…
```
x
```

Out[238…
```
array([0, 1, 2])
```

In [239…
```
y
```

Out[239…
```
array([0, 1, 2])
```

Generating a meshgrid:

In [240…
```
xv, yv = np.meshgrid(x,y)
```

In [241…
```
xv
```

Out[241…
```
array([[0, 1, 2],
       [0, 1, 2],
       [0, 1, 2]])
```

In [242…
```
yv
```

Out[242…
```
array([[0, 0, 0],
       [1, 1, 1],
       [2, 2, 2]])
```

In [243...
```
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 [244...
```
p_1, v_1 = np.meshgrid(p,v)
```

In [245...
```
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 [246...
```
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

In [247...
```
np.meshgrid ([1,2,3], [5,6,7])
```

Out[247...
```
(array([[1, 2, 3],
        [1, 2, 3],
        [1, 2, 3]]),
 array([[5, 5, 5],
        [6, 6, 6],
        [7, 7, 7]]))
```

These arrays, xv and yv, each seperately given the x and y coordinates on a 2D grid. you can do normal numpy operations on these arrays:

In [248...
```
xv**2 + yv**2
```

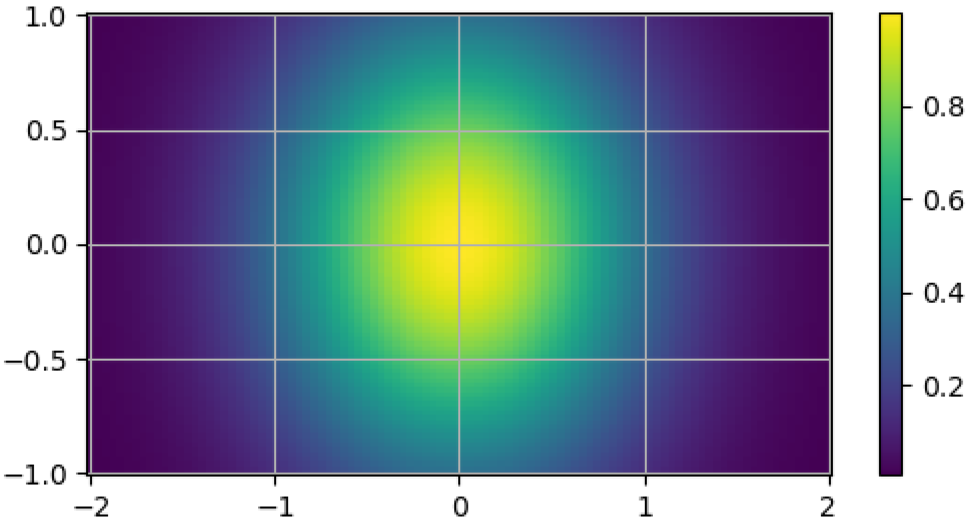
Out[248...
```
array([[0, 1, 4],
       [1, 2, 5],
       [4, 5, 8]])
```

This can be done on a larger scale to plot surface plots of 2D functions Generate functions f(x,y) = e-(x2+y2) for -2<_x<_2 and -1<_y<_1

In [249...
```
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:pcolormsh is typically the preferable function for 2D plotting, as opposed to imshow or pcolor, which take longer.)

In [250...
```
plt.figure(figsize=(6, 3))
plt.pcolormesh(xv,yv,f, shading='auto')
plt.colorbar()
plt.grid()
plt.show()
```



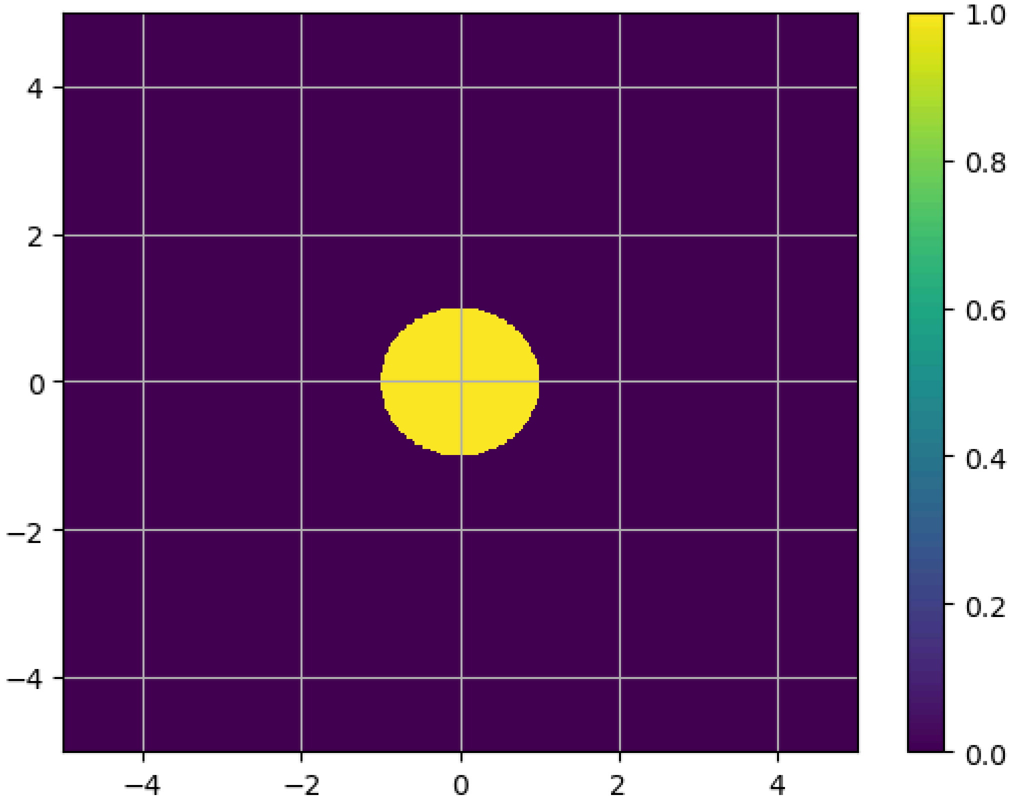
f(x,y) = 1 & x^2+y^2<1\0 & x^2+y^2

In [251...
```
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)
```

In [252...
```
plt.pcolormesh(xv,yv, rectangular_mask,shading='auto')
plt.colorbar()
plt.grid()
plt.show()
```



In [253…
```
# numpy.linspace creates an array of
# 9 linearly placed elements between
# -4 and 4, both inclusive

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

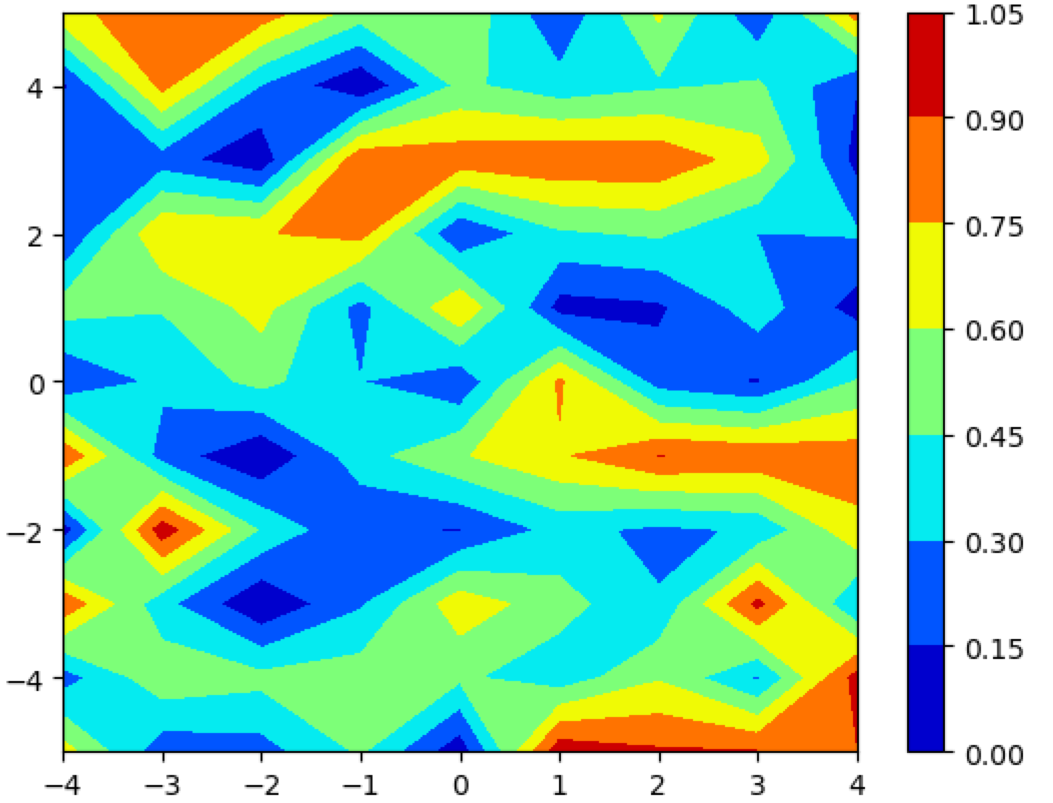
In [254…
```
# numpy.linspace creates an array of
# 9 linearly placed elements between
# -4 and 4, both inclusive
```

In [255…
```
y = np.linspace(-5, 5, 11)
```

In [256…
```
x_1, y_1 = np.meshgrid(x,y)
```

In [257…
```
random_data = np.random.random((11, 9))
plt.contourf(x_1, y_1, random_data, cmap = 'jet')

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

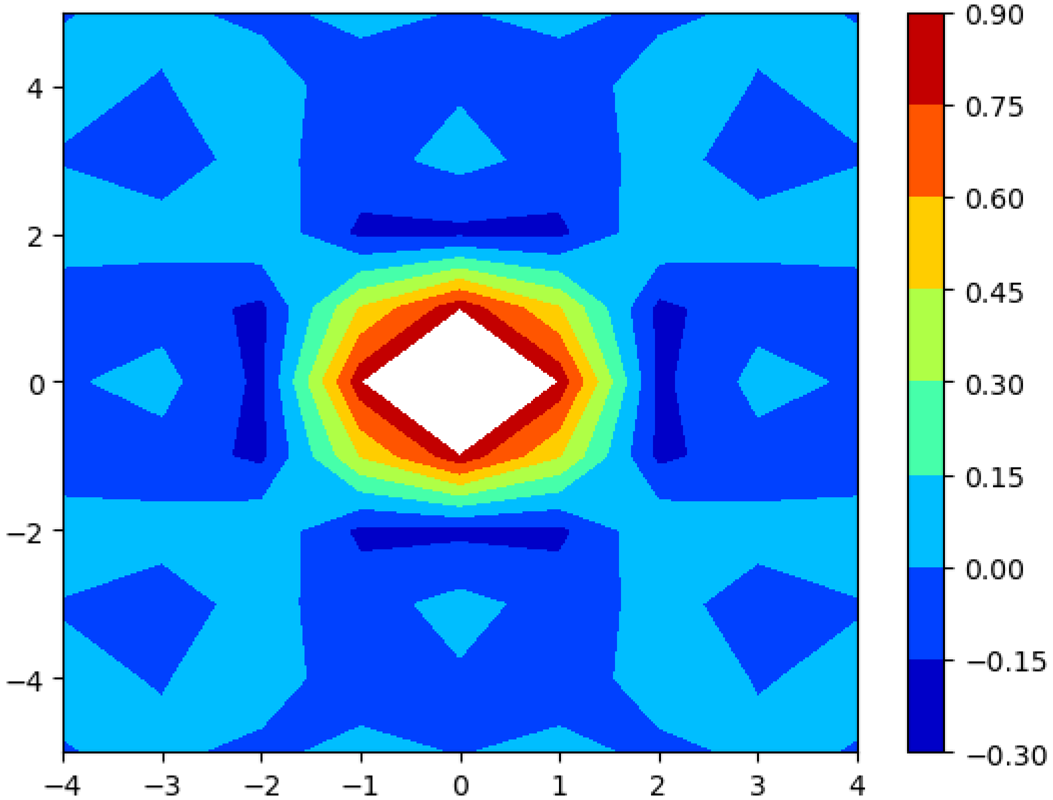


In [258…

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

```
C:\Users\Lenovo\AppData\Local\Temp\ipykernel_18184\174385102.py:1: RuntimeWarnin
g: invalid value encountered in divide
  sine = (np.sin(x_1**2 + y_1**2))/(x_1**2 + y_1**2)
```



In [259…
```
x_1 , y_1 = np.meshgrid(x,y, sparse = True)
```

In [260…
```
x_1
```

Out[260…
```
array([[-4., -3., -2., -1.,  0.,  1.,  2.,  3.,  4.]])
```

In [261…
```
y_1
```

Out[261…
```
array([[-5.],
       [-4.],
       [-3.],
       [-2.],
       [-1.],
       [ 0.],
       [ 1.],
       [ 2.],
       [ 3.],
       [ 4.],
       [ 5.]])
```

np.sort

Return a sorted copy of an array.

In [262…
```
a = np.random.randint(1,100,15) # 1D
a
```

Out[262…
```
array([52, 31, 70, 22, 69, 80, 54, 11, 12, 22, 16, 74,  9, 62,  1],
      dtype=int32)
```

In [263…
```
b = np.random.randint(1,100,24).reshape(6,4) # 2D
b
```

Out[263…
```
array([[14, 98, 60,  4],
       [62, 48, 77, 76],
       [43, 13, 95, 94],
       [ 8, 12, 49, 55],
       [20, 70, 93, 61],
       [29, 29, 53, 35]], dtype=int32)
```

In [264…
```
np.sort(a) # Default = Ascending
```

Out[264…
```
array([ 1,  9, 11, 12, 16, 22, 22, 31, 52, 54, 62, 69, 70, 74, 80],
      dtype=int32)
```

In [265…
```
np.sort(a)[::-1] # Descending order
```

Out[265…
```
array([80, 74, 70, 69, 62, 54, 52, 31, 22, 22, 16, 12, 11,  9,  1],
      dtype=int32)
```

In [266…
```
np.sort(b) # row rise sorting
```

Out[266…
```
array([[ 4, 14, 60, 98],
       [48, 62, 76, 77],
       [13, 43, 94, 95],
       [ 8, 12, 49, 55],
       [20, 61, 70, 93],
       [29, 29, 35, 53]], dtype=int32)
```

In [267…
```
np.sort(b,axis = 0) # column rise sorting
```

Out[267…
```
array([[ 8, 12, 49,  4],
       [14, 13, 53, 35],
       [20, 29, 60, 55],
       [29, 48, 77, 61],
       [43, 70, 93, 76],
       [62, 98, 95, 94]], dtype=int32)
```

np.append

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

In [268…
```
# code
a
```

Out[268…
```
array([52, 31, 70, 22, 69, 80, 54, 11, 12, 22, 16, 74,  9, 62,  1],
      dtype=int32)
```

In [269…
```
np.append(a,200)
```

Out[269…
```
array([ 52,  31,  70,  22,  69,  80,  54,  11,  12,  22,  16,  74,   9,
        62,   1, 200])
```

In [270…
```
b # on 2D
```

Out[270…
```
array([[14, 98, 60,  4],
       [62, 48, 77, 76],
       [43, 13, 95, 94],
       [ 8, 12, 49, 55],
       [20, 70, 93, 61],
       [29, 29, 53, 35]], dtype=int32)
```

In [271…
```
# Adding random number in new column
np.append(b,np.random.random((b.shape[0],1)),axis=1)
```

Out[271…
```
array([[1.40000000e+01, 9.80000000e+01, 6.00000000e+01, 4.00000000e+00,
        6.95161873e-01],
       [6.20000000e+01, 4.80000000e+01, 7.70000000e+01, 7.60000000e+01,
        7.96607018e-02],
       [4.30000000e+01, 1.30000000e+01, 9.50000000e+01, 9.40000000e+01,
        9.64363492e-01],
       [8.00000000e+00, 1.20000000e+01, 4.90000000e+01, 5.50000000e+01,
        7.68240037e-02],
       [2.00000000e+01, 7.00000000e+01, 9.30000000e+01, 6.10000000e+01,
        6.15007453e-02],
       [2.90000000e+01, 2.90000000e+01, 5.30000000e+01, 3.50000000e+01,
        1.33242799e-01]])
```

np.concatenate

numpy.concatenate()funcion concatenate a sequence of arrays along an exiting axis.

In [272…
```
# code
c=np.arange(6).reshape(2,3)
d=np.arange(6,12).reshape(2,3)
```

In [273…
```
c
```

Out[273…
```
array([[0, 1, 2],
       [3, 4, 5]])
```

In [274…
```
d
```

Out[274…
```
array([[ 6,  7,  8],
       [ 9, 10, 11]])
```

we can use it replacement of vstack and hstack

In [275…
```
np.concatenate((c,d)) # Raw wise
```

Out[275…
```
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11]])
```

In [276…
```
np.concatenate((c,d),axis =1) # column wise
```

Out[276…
```
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 from an array given as parameter in np.unique() method.

In [277…
```
# code
e = np.array([1,1,2,2,3,3,4,4,5,5,6,6])
```

In [278…
```
e
```

Out[278…
```
array([1, 1, 2, 2, 3, 3, 4, 4, 5, 5, 6, 6])
```

In [279…
```
np.unique(e)
```

Out[279…
```
array([1, 2, 3, 4, 5, 6])
```

np.expand_dims

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

In [280…
```
# code
a
```

Out[280…
```
array([52, 31, 70, 22, 69, 80, 54, 11, 12, 22, 16, 74,  9, 62,  1],
      dtype=int32)
```

In [281…
```
a.shape # 1 D
```

Out[281…
```
(15,)
```

In [282…
```
# converting into 2D array
np.expand_dims(a,axis = 0).shape # 2D
```

Out[282…
```
(1, 15)
```

In [283…
```
np.expand_dims(a,axis = 1)
```

Out[283…
```
array([[52],
       [31],
       [70],
       [22],
       [69],
       [80],
       [54],
       [11],
       [12],
       [22],
       [16],
       [74],
       [ 9],
       [62],
       [ 1]], dtype=int32)
```

we can use in row vector and column vector. expand_dims() is used to insert an addition dimention in input Tensor.

In [284…
```
np.expand_dims(a,axis = 1).shape
```

Out[284…
```
(15, 1)
```

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

In [285…
```
a
```

Out[285…
```
array([52, 31, 70, 22, 69, 80, 54, 11, 12, 22, 16, 74,  9, 62,  1],
      dtype=int32)
```

In [286…
```
# find all indices with value greater than 50
np.where(a>50)
```

Out[286…
```
(array([ 0,  2,  4,  5,  6, 11, 13]),)
```

np.where(condition,True,false)

In [287…
```
# replace all values > 50 with 0
```

```
np.where(a>50,0,a)
```

Out[287…
```
array([ 0, 31,  0, 22,  0,  0,  0, 11, 12, 22, 16,  0,  9,  0,  1],
      dtype=int32)
```

In [288…
```
# print and replace all even numbers to 0
np.where(a%2 == 0,0,a)
```

Out[288…
```
array([ 0, 31,  0,  0, 69,  0,  0, 11,  0,  0,  0,  0,  9,  0,  1],
      dtype=int32)
```

np.argmax

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

In [289…
```
# code
a
```

Out[289…
```
array([52, 31, 70, 22, 69, 80, 54, 11, 12, 22, 16, 74,  9, 62,  1],
      dtype=int32)
```

In [290…
```
np.argmax(a) # biggest number : index number
```

Out[290…
```
np.int64(5)
```

In [291…
```
b # on 2D
```

Out[291…
```
array([[14, 98, 60,  4],
       [62, 48, 77, 76],
       [43, 13, 95, 94],
       [ 8, 12, 49, 55],
       [20, 70, 93, 61],
       [29, 29, 53, 35]], dtype=int32)
```

In [292…
```
np.argmax(b,axis = 1) # row wise bigest number : index
```

Out[292…
```
array([1, 2, 2, 3, 2, 2])
```

In [293…
```
np.argmax(b,axis = 0) # column wise bigest number : index
```

Out[293…
```
array([1, 0, 2, 2])
```

In [294…
```
# np.argmin

a
```

Out[294…
```
array([52, 31, 70, 22, 69, 80, 54, 11, 12, 22, 16, 74,  9, 62,  1],
      dtype=int32)
```

In [295…
```
np.argmin(a)
```

Out[295…
```
np.int64(14)
```

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 [296…
```
a
```
Out[296…
```
array([52, 31, 70, 22, 69, 80, 54, 11, 12, 22, 16, 74,  9, 62,  1],
      dtype=int32)
```
In [297…
```
np.cumsum(a)
```
Out[297…
```
array([ 52,  83, 153, 175, 244, 324, 378, 389, 401, 423, 439, 513, 522,
        584, 585])
```
In [298…
```
b
```
Out[298…
```
array([[14, 98, 60,  4],
       [62, 48, 77, 76],
       [43, 13, 95, 94],
       [ 8, 12, 49, 55],
       [20, 70, 93, 61],
       [29, 29, 53, 35]], dtype=int32)
```
In [299…
```
np.cumsum(b,axis=0) # column wise calculation or cumulative sum
```
Out[299…
```
array([[ 14,  98,  60,   4],
       [ 76, 146, 137,  80],
       [119, 159, 232, 174],
       [127, 171, 281, 229],
       [147, 241, 374, 290],
       [176, 270, 427, 325]])
```
In [300…
```
# np.cumprod ---> multiply
a
```
Out[300…
```
array([52, 31, 70, 22, 69, 80, 54, 11, 12, 22, 16, 74,  9, 62,  1],
      dtype=int32)
```
In [301…
```
np.cumprod(a)
```
Out[301…
```
array([                 52,                 1612,               112840,
                   2482480,            171291120,          13703289600,
              739977638400,        8139754022400,       97677048268800,
          2148895061913600,    34382320990617600, 2544291753305702400,
       4451881706041769984, -6844953310535235232, -6844953310535235232])
```

np.percentile

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

In [302…
```
a
```

Out[302...
```
array([52, 31, 70, 22, 69, 80, 54, 11, 12, 22, 16, 74,  9, 62,  1],
      dtype=int32)
```

In [303...
```
np.percentile(a,100) # max
```

Out[303...
```
np.float64(80.0)
```

In [304...
```
np.percentile(a,0) # min
```

Out[304...
```
np.float64(1.0)
```

In [305...
```
np.percentile(a,50) # median
```

Out[305...
```
np.float64(31.0)
```

In [306...
```
np.median(a)
```

Out[306...
```
np.float64(31.0)
```

np.histogram

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

In [307...
```
a
```

Out[307...
```
array([52, 31, 70, 22, 69, 80, 54, 11, 12, 22, 16, 74,  9, 62,  1],
      dtype=int32)
```

In [308...
```
np.histogram(a, bins=[10,20,30,40,50,60,70,80,90,100])
```

Out[308...
```
(array([3, 2, 1, 0, 2, 2, 2, 1, 0]),
 array([ 10,  20,  30,  40,  50,  60,  70,  80,  90, 100]))
```

In [309...
```
np.histogram( a, bins = [0,50,100])
```

Out[309...
```
(array([8, 7]), array([  0,  50, 100]))
```

np.corrcoef

Retum pearson product-moment correlation coefficients.

In [310...
```
salary = np.array([20000,40000,25000,35000,60000])
```

In [311...
```
experience = np.array([1,3,2,4,2])
```

In [312...
```
salary
```

Out[312...
```
array([20000, 40000, 25000, 35000, 60000])
```

In [313...
```
experience
```

Out[313...
```
array([1, 3, 2, 4, 2])
```

In [314...
```
np.corrcoef(salary,experience) # correlation coefficient
```

Out[314...
```
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 [315...
```
# code

a
```

Out[315...
```
array([52, 31, 70, 22, 69, 80, 54, 11, 12, 22, 16, 74,  9, 62,  1],
      dtype=int32)
```

In [316...
```
items = [10,20,30,40,50,60,70,80,90,100]
np.isin(a,items)
```

Out[316...
```
array([False, False,  True, False, False,  True, False, False, False,
       False, False, False, False, False, False])
```

In [317...
```
a[np.isin(a,items)]
```

Out[317...
```
array([70, 80], dtype=int32)
```

np.flip

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

In [318...
```
# code

a
```

Out[318...
```
array([52, 31, 70, 22, 69, 80, 54, 11, 12, 22, 16, 74,  9, 62,  1],
      dtype=int32)
```

In [319...
```
np.flip(a) # reverse
```

Out[319...
```
array([ 1, 62,  9, 74, 16, 22, 12, 11, 54, 80, 69, 22, 70, 31, 52],
      dtype=int32)
```

In [320...
```
b
```

Out[320…
```
array([[14, 98, 60,  4],
       [62, 48, 77, 76],
       [43, 13, 95, 94],
       [ 8, 12, 49, 55],
       [20, 70, 93, 61],
       [29, 29, 53, 35]], dtype=int32)
```

In [321…
```
np.flip(b)
```

Out[321…
```
array([[35, 53, 29, 29],
       [61, 93, 70, 20],
       [55, 49, 12,  8],
       [94, 95, 13, 43],
       [76, 77, 48, 62],
       [ 4, 60, 98, 14]], dtype=int32)
```

In [322…
```
np.flip(b,axis = 1) # row
```

Out[322…
```
array([[ 4, 60, 98, 14],
       [76, 77, 48, 62],
       [94, 95, 13, 43],
       [55, 49, 12,  8],
       [61, 93, 70, 20],
       [35, 53, 29, 29]], dtype=int32)
```

In [323…
```
np.flip(b,axis = 0) # column
```

Out[323…
```
array([[29, 29, 53, 35],
       [20, 70, 93, 61],
       [ 8, 12, 49, 55],
       [43, 13, 95, 94],
       [62, 48, 77, 76],
       [14, 98, 60,  4]], dtype=int32)
```

np.put

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

In [324…
```
# code
a
```

Out[324…
```
array([52, 31, 70, 22, 69, 80, 54, 11, 12, 22, 16, 74,  9, 62,  1],
      dtype=int32)
```

In [325…
```
np.put(a,[0,1],[110,530]) # permanent changes
```

In [326…
```
a
```

Out[326…
```
array([110, 530,  70,  22,  69,  80,  54,  11,  12,  22,  16,  74,   9,
        62,   1], dtype=int32)
```

np.delete

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

In [327…
```
# code

a
```

Out[327…
```
array([110, 530,  70,  22,  69,  80,  54,  11,  12,  22,  16,  74,   9,
        62,   1], dtype=int32)
```

In [328…
```
np.delete(a,0) # deleted 0 index items
```

Out[328…
```
array([530,  70,  22,  69,  80,  54,  11,  12,  22,  16,  74,   9,  62,
         1], dtype=int32)
```

In [329…
```
np.delete(a,[0,2,4]) # deleted 0,2,4 index items
```

Out[329…
```
array([530,  22,  80,  54,  11,  12,  22,  16,  74,   9,  62,   1],
      dtype=int32)
```

Set functions

- np.union 1d
- np.interset 1d
- np.setdiff1d
- np.setxort1d
- np.in1d

In [330…
```
m = np.array([1,2,3,4,5])
```

In [331…
```
n = np.array([3,4,5,6,7])
```

In [332…
```
# union
np.union1d(m,n)
```

Out[332…
```
array([1, 2, 3, 4, 5, 6, 7])
```

In [333…
```
# Intersection
np.intersect1d(m,n)
```

Out[333…
```
array([3, 4, 5])
```

In [334…
```
# set difference
np.setdiff1d(m,n)
```

Out[334…
```
array([1, 2])
```

In [335…
```
np.setdiff1d(n,m)
```

Out[335…
```
array([6, 7])
```

In [336…
```
# set xor
np.setxor1d(m,n)
```

Out[336…
```
array([1, 2, 6, 7])
```

In [337…
```
# in 1D ( like membership operator)
np.in1d(m,1)
```

```
C:\Users\Lenovo\AppData\Local\Temp\ipykernel_18184\3636552393.py:2: DeprecationWarning: `in1d` is deprecated. Use `np.isin` instead.
  np.in1d(m,1)
```

Out[337…
```
array([ True, False, False, False, False])
```

In [338…
```
m[np.in1d(m,1)]
```

```
C:\Users\Lenovo\AppData\Local\Temp\ipykernel_18184\524887286.py:1: DeprecationWarning: `in1d` is deprecated. Use `np.isin` instead.
  m[np.in1d(m,1)]
```

Out[338…
```
array([1])
```

In [339…
```
np.in1d(m,10)
```

```
C:\Users\Lenovo\AppData\Local\Temp\ipykernel_18184\3337556509.py:1: DeprecationWarning: `in1d` is deprecated. Use `np.isin` instead.
  np.in1d(m,10)
```

Out[339…
```
array([False, False, False, False, False])
```

np.clip

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

In [340…
```
# code
a
```

Out[340…
```
array([110, 530,  70,  22,  69,  80,  54,  11,  12,  22,  16,  74,   9,
        62,   1], dtype=int32)
```

In [341…
```
np.clip(a,a_min=15, a_max=50)
```

Out[341…
```
array([50, 50, 50, 22, 50, 50, 50, 15, 15, 22, 16, 50, 15, 50, 15],
      dtype=int32)
```

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 array.

In [342…
```
arr = np.array([[1,2,3],[4,5,6]])
swapped_arr = np.swapaxes(arr,0,1)
```

In [343…
```
arr
```

Out[343…
```
array([[1, 2, 3],
       [4, 5, 6]])
```

In [344…
```
swapped_arr
```

Out[344…
```
array([[1, 4],
       [2, 5],
       [3, 6]])
```

In [345…
```
print("original array:")
print(swapped_arr)
```

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

In [346…
```
print("Swapped array:")
print(swapped_arr)
```

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