Numpy Arrays Vs Python Sequences

NumPy arrays have a fixed size at creation, unlike Python lists (which can grow 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-sequence input, they convert such input to NumPy arrays prior to processing, and they often output NumPy arrays.

Speed of List Vs Numpy

List

In [1]:
```
# 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)
```

```
2.0619215965270996
```

Numpy

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

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

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

```
0.1120920181274414
```

In [3]:
```
2.7065064907073975 / 0.02248692512512207
```

Out[3]:
```
120.35911871666826
```

so ,**Numpy** is Faster than Normal Python programming ,we can see in above Example. because Numpy uses C type array

Memory Used for List Vs Numpy

List

In [4]:
```
P = [i for i in range(10000000)]

import sys

sys.getsizeof(P)
```

Out[4]:
```
89095160
```

Numpy

In [5]:
```
R = np.arange(10000000)

sys.getsizeof(R)
```

Out[5]:
```
40000104
```

In [6]:
```
# we can decrease more in numpy

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

sys.getsizeof(R)
```

Out[6]:
```
20000104
```

Advance Indexing and Slicing

In [7]:
```
# Normal Indexing and slicing

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

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

In [8]:
```
# Fetching 5 from array

w[1,2]
```

Out[8]:
```
5
```

In [9]:
```
# Fetching 4,5,7,8
w[1:3]
```

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

In [10]:
```
w[1:3 , 1:3]
```

Out[10]:
```
array([[4, 5],
       [7, 8]])
```

Fancy Indexing

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

In [11]:
```python
w
```

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

In [12]:
```python
# Fetch 1,3,4 row

w[[0,2,3]]
```

Out[12]:
```
array([[ 0,  1,  2],
       [ 6,  7,  8],
       [ 9, 10, 11]])
```

In [13]:
```python
# New array

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

Out[13]:
```
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 [14]:
```python
# Fetch 1, 3, ,4, 6 rows

z[[0,2,3,5]]
```

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

In [15]:
```python
# Fetch 1,3,4  columns

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

Out[15]:
```
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 operations on specific subsets of data.

In [16]:
```
G = np.random.randint(1,100,24).reshape(6,4)
```

In [17]:
```
G
```

Out[17]:
```
array([[64, 51, 75, 50],
       [ 8, 86,  6, 53],
       [60, 50, 49, 95],
       [75, 79, 98, 34],
       [45, 35, 87, 58],
       [56, 26, 93, 17]])
```

In [18]:
```
# find all numbers greater than 50

G > 50
```

Out[18]:
```
array([[ True,  True,  True, False],
       [False,  True, False,  True],
       [ True, False, False,  True],
       [ True,  True,  True, False],
       [False, False,  True,  True],
       [ True, False,  True, False]])
```

In [19]:
```
# Where is True , it gives result , everything other that removed.we got value

G[G > 50]
```

Out[19]:
```
array([64, 51, 75, 86, 53, 60, 95, 75, 79, 98, 87, 58, 56, 93])
```

it is best Techinque to filter the data in given condition

In [20]:
```
# find out even numbers

G % 2 == 0
```

Out[20]:
```
array([[ True, False, False,  True],
       [ True,  True,  True, False],
       [ True,  True, False, False],
       [False, False,  True,  True],
       [False, False, False,  True],
       [ True,  True, False, False]])
```

In [21]:
```
# Gives only  the even numbers

G [ G % 2 == 0]
```

Out[21]:
```
array([64, 50,  8, 86,  6, 60, 50, 98, 34, 58, 56, 26])
```

In [22]:
```
# find all numbers greater than 50 and are even

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

Out[22]:
```
array([[ True, False, False, False],
       [False,  True, False, False],
       [ True, False, False, False],
       [False, False,  True, False],
       [False, False, False,  True],
       [ True, False, False, False]])
```

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

In [23]:
```
# Result

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

Out[23]:
```
array([64, 86, 60, 98, 58, 56])
```

In [24]:
```
# find all numbers not divisible by 7

G % 7 == 0
```

Out[24]:
```
array([[False, False, False, False],
       [False, False, False, False],
       [False, False,  True, False],
       [False, False,  True, False],
       [False,  True, False, False],
       [ True, False, False, False]])
```

In [25]:
```
# Result
G[~(G % 7 == 0)] # (~) = Not
```

Out[25]:
```
array([64, 51, 75, 50,  8, 86,  6, 53, 60, 50, 95, 75, 79, 34, 45, 87, 58,
       26, 93, 17])
```

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 [26]:
```
# 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 [27]:
```
# 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.

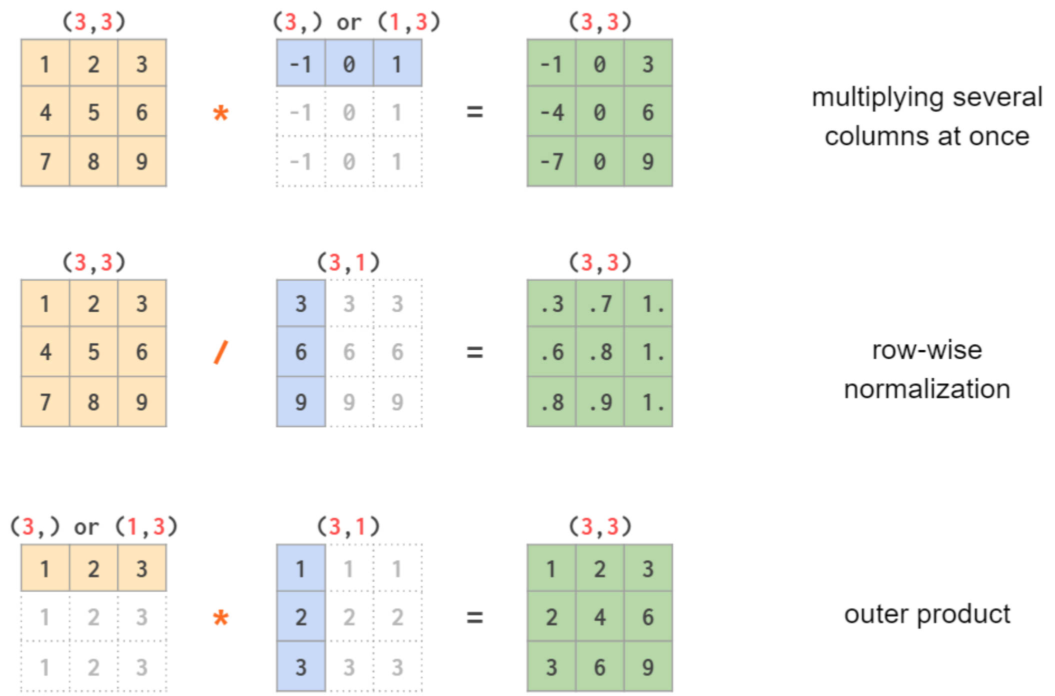
ex : (3,2) = 2D , (3) =1D ---> Convert into (1,3)
(3,3,3) = 3D ,(3) = 1D ---> Convert into (1,1,3)

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

- If the sizes of 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 [28]:
```python
# More examples

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

print(a) # 2 D
```

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

In [29]:
```python
print(b) # 1 D
```

```
[0 1 2]
```

In [30]:
```python
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 [31]:
```
# Could not Broadcast

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

print(a)
print(b)

print(a+b)
```

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

---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
~\AppData\Local\Temp/ipykernel_9360/470058718.py in <module>
      7 print(b)
      8 
----> 9 print(a+b)

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

EXPLANATION : Arthematic 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 equals to b . so it got failed

In [32]:
```
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) convert (1)to 3 than (3,3) . finally it equally.

In [33]:
```
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 [34]:
```
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 [35]:
```
# doesnt work

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)
~\AppData\Local\Temp/ipykernel_9360/1200695402.py in <module>
      7 print(b)
      8 
----> 9 print(a+b)

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

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

In [36]:
```
# Not Work
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)
~\AppData\Local\Temp/ipykernel_9360/2417388683.py in <module>
      6 print(b)
      7 
----> 8 print(a+b)

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

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

Working with mathematical formulas

In [37]:
```
k = np.arange(10)
```

In [38]:
```
k
```

Out[38]:
```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

In [39]:
```
np.sum(k)
```

Out[39]:
```
45
```

In [40]:
```
np.sin(k)
```

Out[40]:
```
array([ 0.        ,  0.84147098,  0.90929743,  0.14112001, -0.7568025 ,
       -0.95892427, -0.2794155 ,  0.6569866 ,  0.98935825,  0.41211849])
```

sigmoid

In [44]:
```
def sigmoid(array):
    return 1/(1+np.exp(-(array)))
k = np.arange(10)
sigmoid(k)
```

Out[44]:
```
array([0.5       , 0.73105858, 0.88079708, 0.95257413, 0.98201379,
       0.99330715, 0.99752738, 0.99908895, 0.99966465, 0.99987661])
```

In [45]:
```
k = np.arange(100)
sigmoid(k)
```

Out[45]:
```
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 [46]:
```
actual = np.random.randint(1,50,25)
predicted = np.random.randint(1,50,25)
```

In [47]:
```
actual
```

Out[47]:
```
array([17,  4,  4, 24, 18, 44, 22, 25, 17, 39,  3, 34, 37, 12, 47, 22, 37,
        9, 47, 38, 27, 46, 47, 34,  8])
```

In [48]:
```
predicted
```

Out[48]:
```
array([47, 31, 30, 17,  7, 22,  1, 16,  1, 24, 16,  7,  6, 37, 18, 15,  2,
       33, 25, 33,  9, 17, 36,  7, 16])
```

In [50]:
```
def mse(actual,predicted):
    return np.mean((actual-predicted)**2)

mse(actual,predicted)
```

Out[50]:
```
469.0
```

In [51]:
```
# detailed

actual-predicted
```

Out[51]:
```
array([-30, -27, -26,   7,  11,  22,  21,   9,  16,  15, -13,  27,  31,
       -25,  29,   7,  35, -24,  22,   5,  18,  29,  11,  27,  -8])
```

In [52]:
```
(actual-predicted)**2
```

Out[52]:
```
array([ 900,  729,  676,   49,  121,  484,  441,   81,  256,  225,  169,
        729,  961,  625,  841,   49, 1225,  576,  484,   25,  324,  841,
        121,  729,   64], dtype=int32)
```

In [53]:
```
np.mean((actual-predicted)**2)
```

Out[53]:
```
469.0
```

Working with Missing Values

In [55]:
```
# Working with missing values -> np.nan

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

Out[55]:
```
array([ 1.,  2.,  3.,  4., nan,  6.])
```

In [56]:
```
np.isnan(S)
```

Out[56]:
```
array([False, False, False, False,  True, False])
```

In [57]:
```
S[np.isnan(S)] # Nan values
```

Out[57]:
```
array([nan])
```

In [58]:
```
S[~np.isnan(S)] # Not Nan Values
```

Out[58]:
```
array([1., 2., 3., 4., 6.])
```

Plotting Graphs

In [59]:
```
# plotting a 2D plot
# x = y

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

Out[59]:
```
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.        ])
```

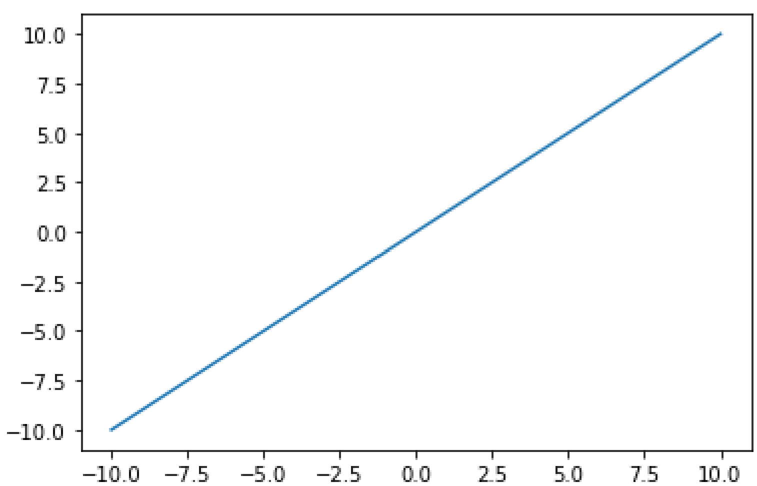
In [60]:
```
y = x
```

In [61]:
```
y
```

Out[61]:
```
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.        ])
```

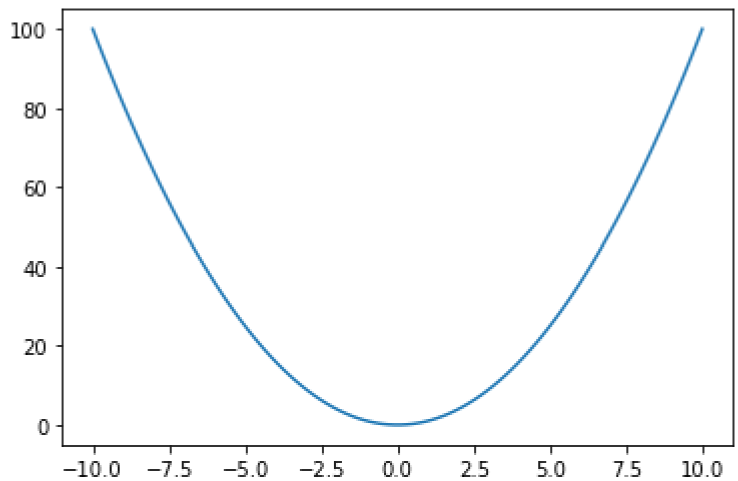
In [62]:
```
import matplotlib.pyplot as plt

plt.plot(x ,y)
```

Out[62]:
```
[<matplotlib.lines.Line2D at 0x1172fe48bb0>]
```



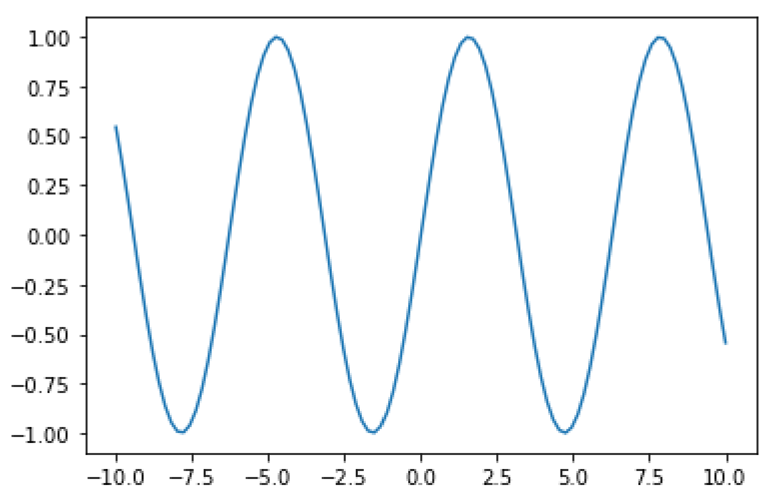
In [63]:
```
# y = x^2

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

plt.plot(x,y)
```

Out[63]: [<matplotlib.lines.Line2D at 0x117324e7310>]



In [64]:
```
# y = sin(x)

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

plt.plot(x,y)
```

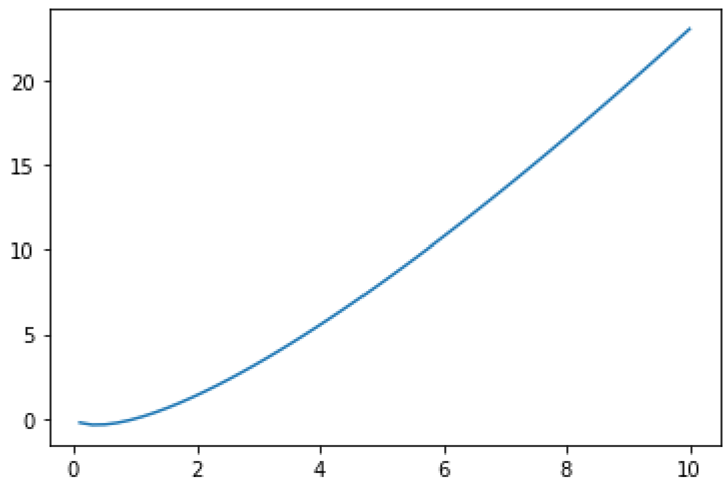
Out[64]: [<matplotlib.lines.Line2D at 0x11732560190>]



In [65]:
```
# y = xlog(x)
x = np.linspace(-10,10,100)
y = x * np.log(x)

plt.plot(x,y)
```

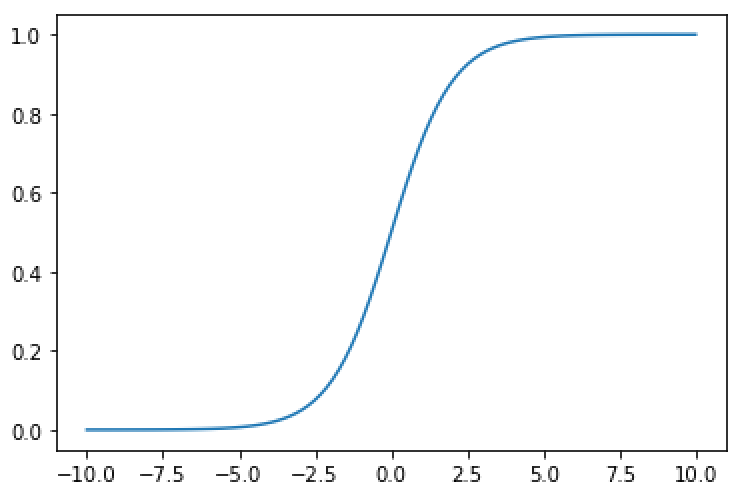
```
C:\Users\user\AppData\Local\Temp/ipykernel_9360/2564014901.py:3: RuntimeWarni
ng: invalid value encountered in log
  y = x * np.log(x)
```

Out[65]: [<matplotlib.lines.Line2D at 0x117325c97f0>]



In [66]:
```
# sigmoid
x = np.linspace(-10,10,100)
y = 1/(1+np.exp(-x))

plt.plot(x,y)
```

Out[66]: [<matplotlib.lines.Line2D at 0x1173262f700>]



In [ ]:

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
```

Meshgrid

Meshgrids 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**

- Creating/Plotting 2D functions f(x,y)
- Generating combinations of 2 or more numbers

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

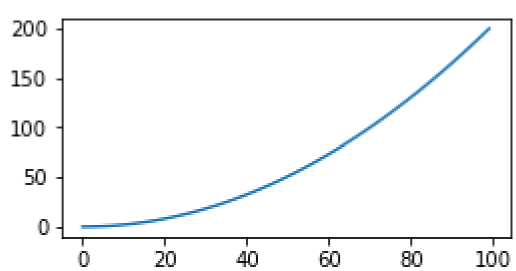
In [2]:
```
x = np.linspace(0,10,100)
y = np.linspace(0,10,100)
```

Try to create 2D function

In [3]:
```
f = x**2+y**2
```

Plot

In [4]:
```
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 [5]:
```
x = np.arange(3)
y = np.arange(3)
```

In [6]:
```
x
```

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

In [7]:
```
y
```

Out[7]:
```
array([0, 1, 2])
```

Generating a meshgrid:

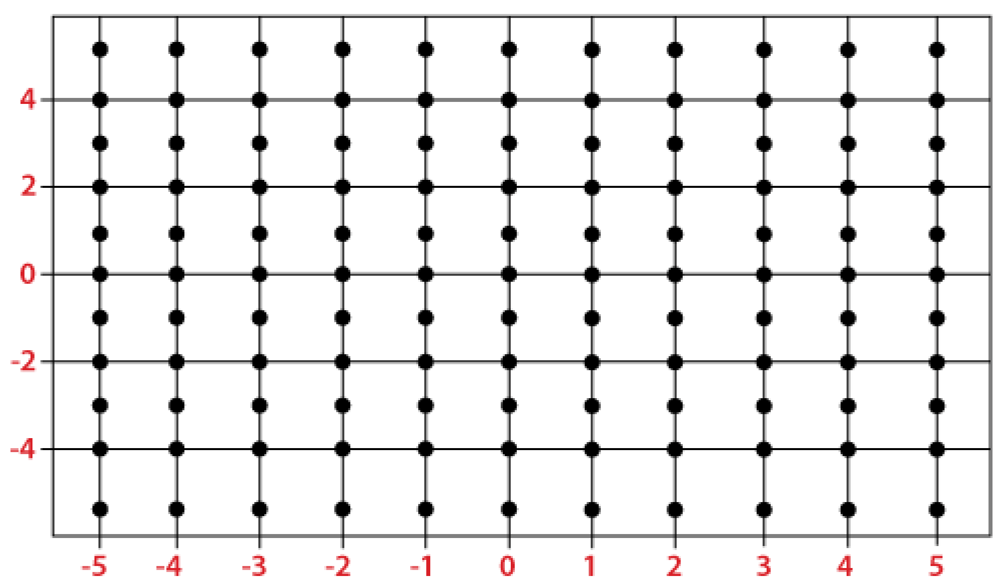
In [8]:
```
xv ,yv = np.meshgrid(x,y)
```

In [9]:
```
xv
```

Out[9]:
```
array([[0, 1, 2],
       [0, 1, 2],
       [0, 1, 2]])
```

In [10]:
```
yv
```

Out[10]:
```
array([[0, 0, 0],
       [1, 1, 1],
       [2, 2, 2]])
```



In [11]:
```
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 [12]:
```
P_1, V_1 = np.meshgrid(P,V)
```

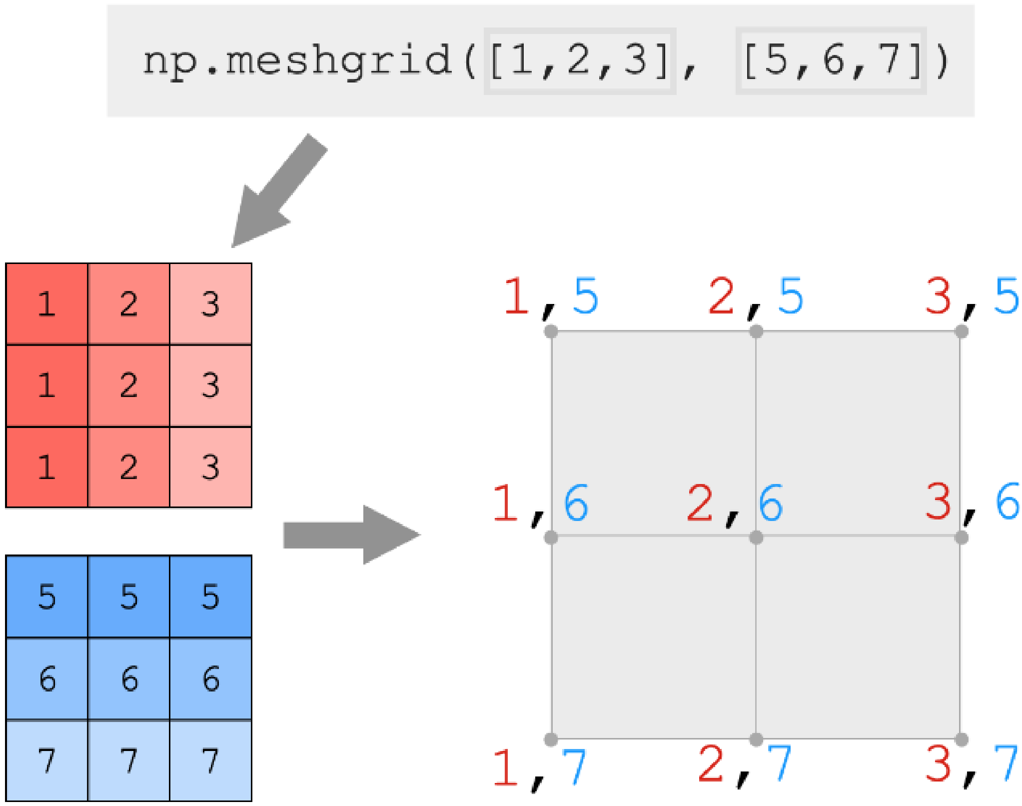
In [13]:
```
 print(P_1)
```

```
[[-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 [14]:
```
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 arrays, xv and yv, each seperately give the x and y coordinates on a 2D grid. You can do normal numpy operations on these arrays:

In [15]:
```
xv**2 + yv**2
```

Out[15]:
```
array([[0, 1, 4],
       [1, 2, 5],
       [4, 5, 8]], dtype=int32)
```

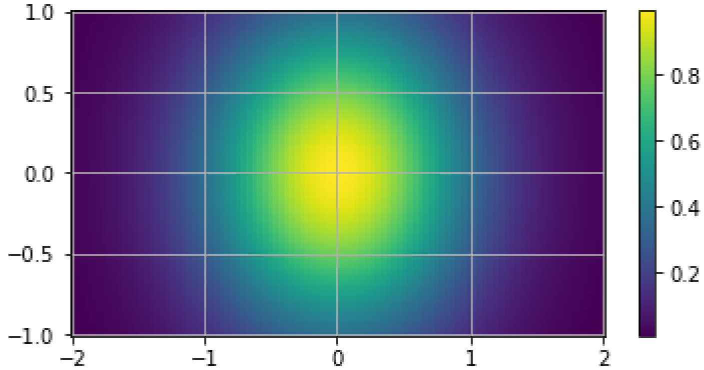
This can be done on a larger scale to plot surface plots of 2D functions

Generate functions **f(x, y) = e−(x²+y²) for −2 ≤ x ≤ 2 and −1 ≤ y ≤1**

In [16]:
```
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)
```

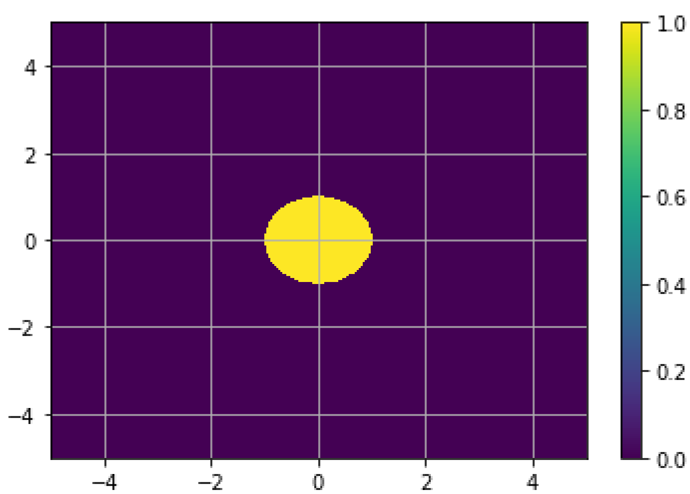
Note: pcolormesh is typically the preferable function for 2D plotting, as opposed to imshow or pcolor, which take longer.)

In [17]:
```
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 [18]:
```
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()
```



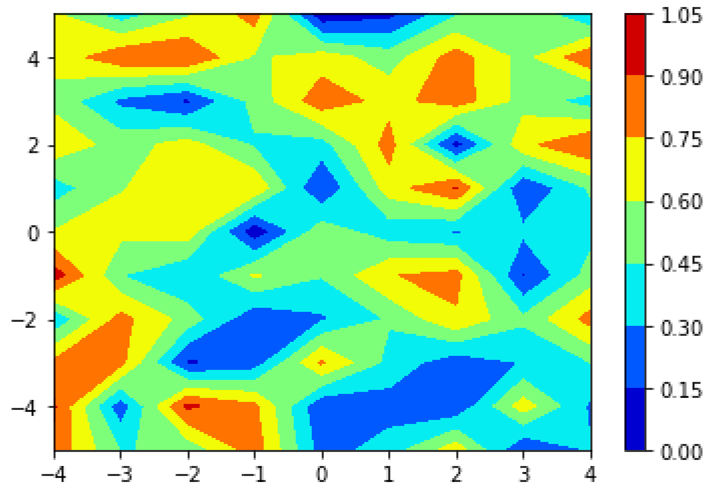
In [19]:
```
# numpy.linspace creates an array of
# 9 linearly placed elements between
# -4 and 4, both inclusive

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

In [20]:
```
# numpy.linspace creates an array of
# 9 linearly placed elements between
# -4 and 4, both inclusive
```

In [21]:
```
y = np.linspace(-5, 5, 11)
```

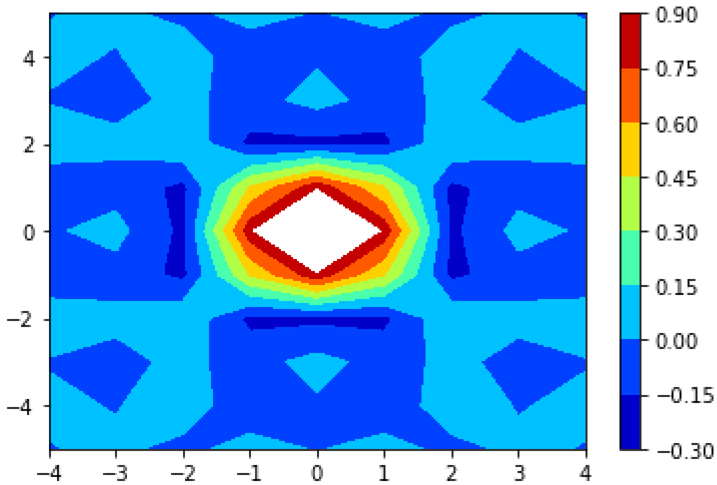
In [22]:
```
x_1, y_1 = np.meshgrid(x, y)
```

In [23]:
```
random_data = np.random.random((11, 9))
plt.contourf(x_1, y_1, random_data, cmap = 'jet')

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



In [24]:
```
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\user\AppData\Local\Temp/ipykernel_3612/3873722910.py:1: RuntimeWarni
ng: invalid value encountered in true_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 [25]:
```
x_1, y_1 = np.meshgrid(x, y, sparse = True)
```

In [26]:
```
x_1
```

Out[26]:
```
array([[-4., -3., -2., -1.,  0.,  1.,  2.,  3.,  4.]])
```

In [27]:
```
y_1
```

Out[27]:
```
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 [28]:
```
a = np.random.randint(1,100,15)  #1D
a
```

Out[28]:
```
array([46, 53, 15, 44, 33, 39, 76, 60, 68, 12, 87, 66, 74, 10, 98])
```

In [31]:
```
b = np.random.randint(1,100,24).reshape(6,4) # 2D
b
```

Out[31]:
```
array([[ 6, 51, 40, 85],
       [35, 28, 91, 68],
       [27, 30,  6,  4],
       [18, 48, 48, 15],
       [35, 45, 99, 17],
       [42, 29, 88, 31]])
```

In [32]:
```
np.sort(a) # Default= Ascending
```

Out[32]:
```
array([10, 12, 15, 33, 39, 44, 46, 53, 60, 66, 68, 74, 76, 87, 98])
```

In [36]:
```
np.sort(a)[::-1] # Descending order
```

Out[36]:
```
array([98, 87, 76, 74, 68, 66, 60, 53, 46, 44, 39, 33, 15, 12, 10])
```

In [33]:
```
np.sort(b) # row rise sorting
```

Out[33]:
```
array([[ 6, 40, 51, 85],
       [28, 35, 68, 91],
       [ 4,  6, 27, 30],
       [15, 18, 48, 48],
       [17, 35, 45, 99],
       [29, 31, 42, 88]])
```

In [35]:
```
np.sort(b,axis = 0) # column rise sorting
```

Out[35]:
```
array([[ 6, 28,  6,  4],
       [18, 29, 40, 15],
       [27, 30, 48, 17],
       [35, 45, 88, 31],
       [35, 48, 91, 68],
       [42, 51, 99, 85]])
```

np.append

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

In [37]:
```
# code

a
```

Out[37]:
```
array([46, 53, 15, 44, 33, 39, 76, 60, 68, 12, 87, 66, 74, 10, 98])
```

In [38]:
```
np.append(a,200)
```

Out[38]:
```
array([ 46,  53,  15,  44,  33,  39,  76,  60,  68,  12,  87,  66,  74,
        10,  98, 200])
```

In [39]:
```
b   # on 2D
```

Out[39]:
```
array([[ 6, 51, 40, 85],
       [35, 28, 91, 68],
       [27, 30,  6,  4],
       [18, 48, 48, 15],
       [35, 45, 99, 17],
       [42, 29, 88, 31]])
```

In [42]:
```
# Adding Extra column :1

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

Out[42]:
```
array([ 6., 51., 40., 85., 35., 28., 91., 68., 27., 30.,  6.,  4., 18.,
       48., 48., 15., 35., 45., 99., 17., 42., 29., 88., 31.,  1.,  1.,
        1.,  1.,  1.,  1.])
```

In [43]:
```
np.append(b,np.ones((b.shape[0],1)),axis=1)
```

Out[43]:
```
array([[ 6., 51., 40., 85.,  1.],
       [35., 28., 91., 68.,  1.],
       [27., 30.,  6.,  4.,  1.],
       [18., 48., 48., 15.,  1.],
       [35., 45., 99., 17.,  1.],
       [42., 29., 88., 31.,  1.]])
```

In [44]:
```
#Adding random numbers in new column

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

Out[44]:
```
array([[ 6.        , 51.        , 40.        , 85.        ,  0.47836639],
       [35.        , 28.        , 91.        , 68.        ,  0.98776768],
       [27.        , 30.        ,  6.        ,  4.        ,  0.55833259],
       [18.        , 48.        , 48.        , 15.        ,  0.7730807 ],
       [35.        , 45.        , 99.        , 17.        ,  0.22512908],
       [42.        , 29.        , 88.        , 31.        ,  0.73795824]])
```

np.concatenate

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

In [45]:
```
# code
c = np.arange(6).reshape(2,3)
d = np.arange(6,12).reshape(2,3)
```

In [46]:
```
c
```

Out[46]:
```
array([[0, 1, 2],
       [3, 4, 5]])
```

In [47]:
```
d
```

Out[47]:
```
array([[ 6,  7,  8],
       [ 9, 10, 11]])
```

we can use it replacement of **vstack and hstack**

In [48]:
```
np.concatenate((c,d)) # Row wise
```

Out[48]:
```
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11]])
```

In [49]:
```
np.concatenate((c,d),axis =1 ) # column wise
```

Out[49]:
```
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 [50]:
```
# code
e = np.array([1,1,2,2,3,3,4,4,5,5,6,6])
```

In [51]:
```
e
```

Out[51]:
```
array([1, 1, 2, 2, 3, 3, 4, 4, 5, 5, 6, 6])
```

In [52]:
```
np.unique(e)
```

Out[52]:
```
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 [53]:
```
#code
a
```

Out[53]:
```
array([46, 53, 15, 44, 33, 39, 76, 60, 68, 12, 87, 66, 74, 10, 98])
```

In [57]:
```
a.shape # 1 D
```

Out[57]:
```
(15,)
```

In [56]:
```
# converting into 2D array

np.expand_dims(a,axis = 0)
```

Out[56]:
```
array([[46, 53, 15, 44, 33, 39, 76, 60, 68, 12, 87, 66, 74, 10, 98]])
```

In [59]:
```
np.expand_dims(a,axis = 0).shape # 2D
```

Out[59]:
```
(1, 15)
```

In [60]:
```
np.expand_dims(a,axis = 1)
```

Out[60]:
```
array([[46],
       [53],
       [15],
       [44],
       [33],
       [39],
       [76],
       [60],
       [68],
       [12],
       [87],
       [66],
       [74],
       [10],
       [98]])
```

We can use in row vector and Column vector .
expand_dims() is used to **insert an addition dimension in input Tensor**.

In [61]:
```
np.expand_dims(a,axis = 1).shape
```

Out[61]:
```
(15, 1)
```

np.where

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

In [62]:
```
a
```

Out[62]:
```
array([46, 53, 15, 44, 33, 39, 76, 60, 68, 12, 87, 66, 74, 10, 98])
```

In [63]:
```
# find all indices with value greater than 50

np.where(a>50)
```

Out[63]:
```
(array([ 1,  6,  7,  8, 10, 11, 12, 14], dtype=int64),)
```

np.where(condition, True , false)

In [64]:
```
# replace all values > 50 with 0

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

Out[64]:
```
array([46,  0, 15, 44, 33, 39,  0,  0,  0, 12,  0,  0,  0, 10,  0])
```

In [67]:
```
# print and replace all even numbers to 0

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

Out[67]:
```
array([ 0, 53, 15,  0, 33, 39,  0,  0,  0,  0, 87,  0,  0,  0,  0])
```

np.argmax

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

arg = argument

In [68]:
```
# code
a
```

Out[68]:
```
array([46, 53, 15, 44, 33, 39, 76, 60, 68, 12, 87, 66, 74, 10, 98])
```

In [69]:
```
np.argmax(a) # biggest number : index number
```

Out[69]:
```
14
```

In [71]:
```
b # on 2D
```

Out[71]:
```
array([[ 6, 51, 40, 85],
       [35, 28, 91, 68],
       [27, 30,  6,  4],
       [18, 48, 48, 15],
       [35, 45, 99, 17],
       [42, 29, 88, 31]])
```

In [72]:
```
np.argmax(b,axis =1) # row wise bigest number : index
```

Out[72]:
```
array([3, 2, 1, 1, 2, 2], dtype=int64)
```

In [73]:
```
np.argmax(b,axis =0) # column wise bigest number : index
```

Out[73]:
```
array([5, 0, 4, 0], dtype=int64)
```

In [75]:
```
# np.argmin

a
```

Out[75]:
```
array([46, 53, 15, 44, 33, 39, 76, 60, 68, 12, 87, 66, 74, 10, 98])
```

In [76]:
```
np.argmin(a)
```

Out[76]:
```
13
```

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 [77]:
```
a
```

Out[77]:
```
array([46, 53, 15, 44, 33, 39, 76, 60, 68, 12, 87, 66, 74, 10, 98])
```

In [79]:
```
np.cumsum(a)
```

Out[79]:
```
array([ 46,  99, 114, 158, 191, 230, 306, 366, 434, 446, 533, 599, 673,
       683, 781], dtype=int32)
```

In [85]:
```
b
```

Out[85]:
```
array([[ 6, 51, 40, 85],
       [35, 28, 91, 68],
       [27, 30,  6,  4],
       [18, 48, 48, 15],
       [35, 45, 99, 17],
       [42, 29, 88, 31]])
```

In [86]:
```
np.cumsum(b)
```

Out[86]:
```
array([  6,  57,  97, 182, 217, 245, 336, 404, 431, 461, 467, 471, 489,
       537, 585, 600, 635, 680, 779, 796, 838, 867, 955, 986], dtype=int32)
```

In [84]:
```
np.cumsum(b,axis=1) # row wise calculation or cumulative sum
```

Out[84]:
```
array([[  6,  57,  97, 182],
       [ 35,  63, 154, 222],
       [ 27,  57,  63,  67],
       [ 18,  66, 114, 129],
       [ 35,  80, 179, 196],
       [ 42,  71, 159, 190]], dtype=int32)
```

In [87]:
```
np.cumsum(b,axis=0) # column wise calculation or cumulative sum
```

Out[87]:
```
array([[  6,  51,  40,  85],
       [ 41,  79, 131, 153],
       [ 68, 109, 137, 157],
       [ 86, 157, 185, 172],
       [121, 202, 284, 189],
       [163, 231, 372, 220]], dtype=int32)
```

In [88]:
```
# np.cumprod --> Multiply

a
```

Out[88]:
```
array([46, 53, 15, 44, 33, 39, 76, 60, 68, 12, 87, 66, 74, 10, 98])
```

In [89]:
```
np.cumprod(a)
```

Out[89]:
```
array([         46,        2438,       36570,     1609080,    53099640,
        2070885960, -1526456992, -1393106304,  -241948160,  1391589376,
         809191424,  1867026432,   721002496, -1379909632, -2087157760],
      dtype=int32)
```

np.percentile

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

In [90]:
```
a
```

Out[90]:
```
array([46, 53, 15, 44, 33, 39, 76, 60, 68, 12, 87, 66, 74, 10, 98])
```

In [91]:
```
np.percentile(a,100) # Max
```

Out[91]:
```
98.0
```

In [92]:
```
np.percentile(a,0) # Min
```

Out[92]:
```
10.0
```

In [93]:
```
np.percentile(a,50) # Median
```

Out[93]:
```
53.0
```

In [94]:
```
np.median(a)
```

Out[94]:
```
53.0
```

np.histogram

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

In [95]:
```
a
```

Out[95]:
```
array([46, 53, 15, 44, 33, 39, 76, 60, 68, 12, 87, 66, 74, 10, 98])
```

In [98]:
```
np.histogram(a , bins= [10,20,30,40,50,60,70,80,90,100])
```

Out[98]:
```
(array([3, 0, 2, 2, 1, 3, 2, 1, 1], dtype=int64),
 array([ 10,  20,  30,  40,  50,  60,  70,  80,  90, 100]))
```

In [99]:
```
np.histogram(a , bins= [0,50,100])
```

Out[99]:
```
(array([7, 8], dtype=int64), array([  0,  50, 100]))
```

np.corrcoef

Return Pearson product-moment correlation coefficients.

In [101]:
```
salary = np.array([20000,40000,25000,35000,60000])
experience = np.array([1,3,2,4,2])
```

In [102]:
```
salary
```

Out[102]:
```
array([20000, 40000, 25000, 35000, 60000])
```

In [103]:
```
experience
```

Out[103]:
```
array([1, 3, 2, 4, 2])
```

In [104]:
```
np.corrcoef(salary,experience) # Correlation Coefficient
```

Out[104]:
```
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 [105]:
```
# code

a
```

Out[105]:
```
array([46, 53, 15, 44, 33, 39, 76, 60, 68, 12, 87, 66, 74, 10, 98])
```

In [107]:
```
items = [10,20,30,40,50,60,70,80,90,100]

np.isin(a,items)
```

Out[107]:
```
array([False, False, False, False, False, False, False,  True, False,
       False, False, False, False,  True, False])
```

In [108]:
```
a[np.isin(a,items)]
```

Out[108]:
```
array([60, 10])
```

np.flip

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

In [109]:
```
# code

a
```

Out[109]:
```
array([46, 53, 15, 44, 33, 39, 76, 60, 68, 12, 87, 66, 74, 10, 98])
```

In [110]:
```
np.flip(a) # reverse
```

Out[110]:
```
array([98, 10, 74, 66, 87, 12, 68, 60, 76, 39, 33, 44, 15, 53, 46])
```

In [111]:
```
b
```

Out[111]:
```
array([[ 6, 51, 40, 85],
       [35, 28, 91, 68],
       [27, 30,  6,  4],
       [18, 48, 48, 15],
       [35, 45, 99, 17],
       [42, 29, 88, 31]])
```

In [112]:
```
np.flip(b)
```

Out[112]:
```
array([[31, 88, 29, 42],
       [17, 99, 45, 35],
       [15, 48, 48, 18],
       [ 4,  6, 30, 27],
       [68, 91, 28, 35],
       [85, 40, 51,  6]])
```

In [113]:
```
np.flip(b,axis = 1) # row
```

Out[113]:
```
array([[85, 40, 51,  6],
       [68, 91, 28, 35],
       [ 4,  6, 30, 27],
       [15, 48, 48, 18],
       [17, 99, 45, 35],
       [31, 88, 29, 42]])
```

In [114]:
```
np.flip(b,axis = 0 ) # column
```

Out[114]:
```
array([[42, 29, 88, 31],
       [35, 45, 99, 17],
       [18, 48, 48, 15],
       [27, 30,  6,  4],
       [35, 28, 91, 68],
       [ 6, 51, 40, 85]])
```

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 [115]:
```
# code

a
```

Out[115]:
```
array([46, 53, 15, 44, 33, 39, 76, 60, 68, 12, 87, 66, 74, 10, 98])
```

In [116]:
```
np.put(a,[0,1],[110,530]) # permanent changes
```

In [117]:
```
a
```

Out[117]:
```
array([110, 530,  15,  44,  33,  39,  76,  60,  68,  12,  87,  66,  74,
        10,  98])
```

np.delete

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

In [118]:
```
# code

a
```

Out[118]:
```
array([110, 530,  15,  44,  33,  39,  76,  60,  68,  12,  87,  66,  74,
        10,  98])
```

In [119]:
```
np.delete(a,0) # deleted 0 index item
```

Out[119]:
```
array([530,  15,  44,  33,  39,  76,  60,  68,  12,  87,  66,  74,  10,
        98])
```

In [120]:
```
np.delete(a,[0,2,4]) # deleted 0,2,4 index items
```

Out[120]:
```
array([530,  44,  39,  76,  60,  68,  12,  87,  66,  74,  10,  98])
```

Set functions

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

In [121]:
```
m = np.array([1,2,3,4,5])
n = np.array([3,4,5,6,7])
```

In [122]:
```
# Union

np.union1d(m,n)
```

Out[122]:
```
array([1, 2, 3, 4, 5, 6, 7])
```

In [123]:
```
# Intersection

np.intersect1d(m,n)
```

Out[123]:
```
array([3, 4, 5])
```

In [126]:
```
# Set difference

np.setdiff1d(m,n)
```

Out[126]:
```
array([1, 2])
```

In [127]:
```
np.setdiff1d(n,m)
```

Out[127]:
```
array([6, 7])
```

In [128]:
```
# set Xor

np.setxor1d(m,n)
```

Out[128]:
```
array([1, 2, 6, 7])
```

In [129]:
```
# in 1D ( like membership operator)

np.in1d(m,1)
```

Out[129]:
```
array([ True, False, False, False, False])
```

In [131]:
```
m[np.in1d(m,1)]
```

Out[131]:
```
array([1])
```

In [130]:
```
np.in1d(m,10)
```

Out[130]:
```
array([False, False, False, False, False])
```

np.clip

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

In [132]:
```
# code

a
```

Out[132]:
```
array([110, 530,  15,  44,  33,  39,  76,  60,  68,  12,  87,  66,  74,
        10,  98])
```

In [133]:
```
np.clip(a, a_min=15 , a_max =50)
```

Out[133]:
```
array([50, 50, 15, 44, 33, 39, 50, 50, 50, 15, 50, 50, 50, 15, 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 array.

In [137]:
```
arr = np.array([[1, 2, 3], [4, 5, 6]])
swapped_arr = np.swapaxes(arr, 0, 1)
```

In [138]:
```
arr
```

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

In [139]:
```
swapped_arr
```

Out[139]:
```
array([[1, 4],
       [2, 5],
       [3, 6]])
```

In [140]:
```
print("Original array:")
print(arr)
```

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

In [141]:
```
print("Swapped array:")
print(swapped_arr)
```

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

In [ ]: