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

In [2]:
```
arr1d = np.array([1,2,3,4]) # its a 1d array
print(arr1d)
```

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

In [3]:
```
arr1d.ndim
```

Out[3]:
```
1
```

In [4]:
```
arr1d.ndim # dimension of the array
```

Out[4]:
```
1
```

In [5]:
```
arr1d.size #specifies the size
```

Out[5]:
```
4
```

In [6]:
```
arr2d=np.array([[1,2,3,4],[5,6,7,8]]) #2D array list of list
print(arr2d)
```

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

In [7]:
```
arr2d.ndim
```

Out[7]:
```
2
```

In [8]:
```
arr2d.size #no.of elements in the array
```

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

In [9]:
```
arr2d.shape #returns tuple of row,column)
```

Out[9]:
```
(2, 4)
```

In [10]:
```
arr2d.dtype
```

Out[10]:
```
dtype('int32')
```

In [11]:
```
arr2d=np.array([[1,2,3,4],[5,6,7,8]],dtype=complex) # returns complex form
print(arr2d)
```

```
[[1.+0.j 2.+0.j 3.+0.j 4.+0.j]
 [5.+0.j 6.+0.j 7.+0.j 8.+0.j]]
```

In [12]:
```
arr2d=np.array([[1,2,3,4],[5,6,7,8]],dtype=float) # returns float form
print(arr2d)
```

```
[[1. 2. 3. 4.]
 [5. 6. 7. 8.]]
```

In [13]:
```
#Creating maxtrix of 1's (3*3 MATRIX)
mx_1s = np.array([[1,1,1],[1,1,1],[1,1,1]])
print(mx_1s)
```

```
[[1 1 1]
 [1 1 1]
 [1 1 1]]
```

In [14]:
```
#CREATING ONES MATRIX TO OUR DESIRE ROWS AND COLUMNS
mx=np.ones((4,4)) #DEFAULTY RETURNS FLOATING POINT NUMBER
print(mx)
```

```
[[1. 1. 1. 1.]
 [1. 1. 1. 1.]
 [1. 1. 1. 1.]
 [1. 1. 1. 1.]]
```

In [15]:
```
mx=np.ones((4,4),dtype=int) #TYPECASTING
print(mx)
```

```
[[1 1 1 1]
 [1 1 1 1]
 [1 1 1 1]
 [1 1 1 1]]
```

In [16]:
```
mx=np.ones((4,4),dtype=complex) #TYPECASTING
print(mx)
```

```
[[1.+0.j 1.+0.j 1.+0.j 1.+0.j]
 [1.+0.j 1.+0.j 1.+0.j 1.+0.j]
 [1.+0.j 1.+0.j 1.+0.j 1.+0.j]
 [1.+0.j 1.+0.j 1.+0.j 1.+0.j]]
```

In [17]:
```
#CREATING ZEROS MATRIX TO OUR DESIRE ROWS AND COLUMNS
mx=np.zeros((4,4)) #DEFAULTY RETURNS FLOATING POINT NUMBER
print(mx)
```

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

In [18]:
```
mx=np.zeros((4,4),dtype=int) #TYPECASTING
print(mx)
```

```
[[0 0 0 0]
 [0 0 0 0]
 [0 0 0 0]
 [0 0 0 0]]
```

In [19]:
```
mx=np.zeros((4,4),dtype=bool) #TYPECASTING
print(mx)
```

```
[[False False False False]
 [False False False False]
 [False False False False]
 [lse False]]
```

In [20]:
```
mx=np.zeros((4,4),dtype=str) #TYPECASTING
print(mx)
```

```
[['' '' '' '']
 ['' '' '' '']
 ['' '' '' '']
 ['' '' '' '']]
```

In [21]:
```
#EMPTY MATRIX -> ITS NOT EMPTY IT CAN HAVE ANY NUMBERS

mx=np.empty((3,3))
print(mx)
```

```
[[0.00000000e+000 0.00000000e+000 0.00000000e+000]
 [0.00000000e+000 0.00000000e+000 6.71929278e-321]
 [0.00000000e+000 0.00000000e+000 1.60216183e-306]]
```

Numpy Functions

arange(START,END-1,STEPS,DTYPE): GENERATES 1-D ARRAY

In [22]:
```
arr=np.arange(1,12)
print(arr)
```

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

In [23]:
```
arr3=np.arange(1,13,2)
print(arr3)
```

```
[ 1  3  5  7  9 11]
```

linspace(start, stop, num, endpoint, retstep, dtype): This function is similar to arange() function. In this function, instead of step size, the number of evenly spaced values between the interval is specified

In [24]:
```
x = np.linspace(10,20,5)
print (x)
```

```
[10.  12.5 15.  17.5 20. ]
```

reshape(): coverts 1D to 2D and 2D to 3D

In [25]:
```
arr1d=np.arange(12)
print(arr1d)
```

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

In [26]:
```
arr3d=arr1d.reshape(2,3,2)#(2*3*2=12 so the dimensions should be equal to the size of 1D
print(arr3d)
```

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

 [[ 6  7]
  [ 8  9]
  [10 11]]]
```

In [27]:
```
arr=np.arange(1,13).reshape(2,6)
print(arr)
```

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

ravel(): CONVERTS 2D,3D TO 1D ARRAY

In [28]:
```
arr.ravel()
```

Out[28]:
```
array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12])
```

flatten(): CONVERTS 2D,3D TO 1D ARRAY

In [29]:
```
arr.flatten()
```

Out[29]:
```
array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12])
```

transpose(): converts rows to columns

In [30]:
```
arr.transpose()
```

Out[30]:
```
array([[ 1,  7],
       [ 2,  8],
       [ 3,  9],
       [ 4, 10],
       [ 5, 11],
       [ 6, 12]])
```

In [31]:
```
arr.T
```

Out[31]:
```
array([[ 1,  7],
       [ 2,  8],
       [ 3,  9],
       [ 4, 10],
       [ 5, 11],
       [ 6, 12]])
```

Mathematical Operations

In [32]:
```
arr1=np.arange(1,10).reshape(3,3)
,10).reshape(3,3)
```

```
print(arr1)
print(arr2)
```

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

In [33]:
```
arr1+arr2 #addition of matrix
```

Out[33]:
```
array([[ 2,  4,  6],
       [ 8, 10, 12],
       [14, 16, 18]])
```

In [34]:
```
np.add(arr1,arr2)
```

Out[34]:
```
array([[ 2,  4,  6],
       [ 8, 10, 12],
       [14, 16, 18]])
```

In [35]:
```
np.subtract(arr1,arr2)
```

Out[35]:
```
array([[0, 0, 0],
       [0, 0, 0],
       [0, 0, 0]])
```

In [36]:
```
np.divide(arr1,arr2)
```

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

In [37]:
```
arr1//arr2
```

Out[37]:
```
array([[1, 1, 1],
       [1, 1, 1],
       [1, 1, 1]], dtype=int32)
```

In [38]:
```
np.multiply(arr1,arr2) #NOT ROW AND COLUMN :-> ONLY ELEMENT WISE
```

Out[38]:
```
array([[ 1,  4,  9],
       [16, 25, 36],
       [49, 64, 81]])
```

In [39]:
```
arr1@arr2 #product of row with column
```

Out[39]:
```
array([[ 30,  36,  42],
       [ 66,  81,  96],
       [102, 126, 150]])
```

In [40]:
```
arr1.dot(arr2) #product of row with column
```

Out[40]:
```
array([[ 30,  36,  42],
       [ 66,  81,  96],
       [102, 126, 150]])
```

Out[41]:
```
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

In [42]:
```
arr1.max()
```

Out[42]:
```
9
```

In [43]:
```
arr1.min()
```

Out[43]:
```
1
```

In [44]:
```
arr1.argmax() # shows the index of max element in the matrix
```

Out[44]:
```
8
```

In [45]:
```
arr1.max(axis = 0) # returns max elements of each columns
                   # 0 -> FOR COLUMNS AND 1 -> FOR ROWS
```

Out[45]:
```
array([7, 8, 9])
```

In [46]:
```
arr1.max(axis = 1)
```

Out[46]:
```
array([3, 6, 9])
```

In [47]:
```
arr1.argmin()
```

Out[47]:
```
0
```

In [48]:
```
arr1.min(axis = 0)
```

Out[48]:
```
array([1, 2, 3])
```

In [49]:
```
arr1.min(axis = 1)
```

Out[49]:
```
array([1, 4, 7])
```

In [50]:
```
np.sum(arr1) # SUM OF ALL ELEMENTS OF A MATRIX
```

Out[50]:
```
45
```

In [51]:
```
np.sum(arr1,axis=0) #SUM OF ELEMENTS OF COLUMNS
```

Out[51]:
```
array([12, 15, 18])
```

In [52]:
```
np.sum(arr1,axis=1)#SUM OF ELEMENTS OF ROW
```

Out[52]:
```
array([ 6, 15, 24])
```

In [53]:
```
np.mean(arr1,dtype=int) #MEAN OF ELEMENTS
```

Out[53]:
```
5
```

In [54]:
```
np.sqrt(arr1) #SQ ROOT OF EACH ELEMENT OF MATRIX
```

Out[54]:
```
array([[1.        , 1.41421356, 1.73205081],
       [2.        , 2.23606798, 2.44948974],
       [2.64575131, 2.82842712, 3.        ]])
```

In [55]:
```
np.std(arr1) #STANDARD DEVIATION
```

Out[55]:
```
2.581988897471611
```

In [56]:
```
np.exp(arr1) # exponenential form......e**x
```

Out[56]:
```
array([[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]])
```

In [57]:
```
np.log(arr1)
```

Out[57]:
```
array([[0.        , 0.69314718, 1.09861229],
       [1.38629436, 1.60943791, 1.79175947],
       [1.94591015, 2.07944154, 2.19722458]])
```

In [58]:
```
np.log10(arr1)
```

Out[58]:
```
array([[0.        , 0.30103   , 0.47712125],
       [0.60205999, 0.69897   , 0.77815125],
       [0.84509804, 0.90308999, 0.95424251]])
```

Python NumPy Array Slicing:

In [59]:
```
arr2=np.arange(1,101).reshape(10,10)
print(arr2)
```

```
[[  1   2   3   4   5   6   7   8   9  10]
 [ 11  12  13  14  15  16  17  18  19  20]
 [ 21  22  23  24  25  26  27  28  29  30]
 [ 31  32  33  34  35  36  37  38  39  40]
 [ 41  42  43  44  45  46  47  48  49  50]
 [ 51  52  53  54  55  56  57  58  59  60]
 [ 61  62  63  64  65  66  67  68  69  70]
 [ 71  72  73  74  75  76  77  78  79  80]
 [ 81  82  83  84  85  86  87  88  89  90]
 [ 91  92  93  94  95  96  97  98  99 100]]
```

In [60]:
```
arr[0,0] #Slicing starts with 0 indexing means 1st element is at 0th row and 0th cloumn
```

Out[60]:
```
1
```

In [61]:
```
arr[0,1]
```

Out[61]:
```
2
```

In [62]:
```
arr2[0,1].ndim #dimensionless
```

Out[62]:
```
0
```

In [63]:
```
arr2[0] # accesing rows
```

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

In [64]:
```
arr2[0].ndim
```

Out[64]:
```
1
```

In [65]:
```
arr2[4,4]
```

Out[65]:
```
45
```

In [66]:
```
arr2[:,0] #accesing columns
```

Out[66]:
```
array([ 1, 11, 21, 31, 41, 51, 61, 71, 81, 91])
```

In [67]:
```
arr2[:,1]
```

Out[67]:
```
array([ 2, 12, 22, 32, 42, 52, 62, 72, 82, 92])
```

In [68]:
```
arr2[...,4] #ALTERNATIVE WAY OF ACCESING COLUMNS

#Slicing can also include ellipsis (…) to make a selection tuple of the same length as th
```

Out[68]:
```
array([ 5, 15, 25, 35, 45, 55, 65, 75, 85, 95])
```

In [69]:
```
arr2[4,...] #ALTERNATIVE WAY OF ACCESING ROWS
```

Out[69]:
```
array([41, 42, 43, 44, 45, 46, 47, 48, 49, 50])
```

In [70]:
```
arr2[...,0:0]
```

Out[70]:
```
array([], shape=(10, 0), dtype=int32)
```

In [71]:
```
arr2[...,0:1] # ACCESSING COLUMNS IN 2-D
```

Out[71]:
```
array([[ 1],
       [11],
       [21],
       [31],
       [41],
       [51],
       [61],
       [71],
```

```
       [81],
       [91]])
```

In [72]:
```
arr2[0:1,...] # ACCESSING ROWS IN 2-D
```

Out[72]:
```
array([[ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10]])
```

In [73]:
```
arr2[...,0:1].ndim
```

Out[73]:
```
2
```

In [74]:
```
arr2[1:4,1:4] # ACCESSING ROWS AND COLUMNS OF A MATRIX
```

Out[74]:
```
array([[12, 13, 14],
       [22, 23, 24],
       [32, 33, 34]])
```

In [75]:
```
arr2[:,1:3]
```

Out[75]:
```
array([[ 2,  3],
       [12, 13],
       [22, 23],
       [32, 33],
       [42, 43],
       [52, 53],
       [62, 63],
       [72, 73],
       [82, 83],
       [92, 93]])
```

In [76]:
```
arr2.itemsize # 4 bytes  space required to store item of elements
```

Out[76]:
```
4
```

Array Concatination and Split

In [77]:
```
arr3=np.arange(1,17).reshape(4,4)
print(arr3)
```

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

In [78]:
```
arr4=np.arange(17,33).reshape(4,4)
print(arr4)
```

```
[[17 18 19 20]
 [21 22 23 24]
 [25 26 27 28]
 [29 30 31 32]]
```

In [79]:
```
np.concatenate((arr3,arr4)) # Pass argument as Tuple : ITS COLUMN WISE CONCATINATION
```

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

```
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12],
       [13, 14, 15, 16],
       [17, 18, 19, 20],
       [21, 22, 23, 24],
       [25, 26, 27, 28],
       [29, 30, 31, 32]])
```

In [80]:
```
np.concatenate((arr3,arr4),axis=1) #ROW wise Concatenation
```

Out[80]:
```
array([[ 1,  2,  3,  4, 17, 18, 19, 20],
       [ 5,  6,  7,  8, 21, 22, 23, 24],
       [ 9, 10, 11, 12, 25, 26, 27, 28],
       [13, 14, 15, 16, 29, 30, 31, 32]])
```

In [81]:
```
np.vstack((arr3,arr4))
```

Out[81]:
```
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12],
       [13, 14, 15, 16],
       [17, 18, 19, 20],
       [21, 22, 23, 24],
       [25, 26, 27, 28],
       [29, 30, 31, 32]])
```

In [82]:
```
np.hstack((arr3,arr4)) # To Concatenate arrays should be of same dimensions only
```

Out[82]:
```
array([[ 1,  2,  3,  4, 17, 18, 19, 20],
       [ 5,  6,  7,  8, 21, 22, 23, 24],
       [ 9, 10, 11, 12, 25, 26, 27, 28],
       [13, 14, 15, 16, 29, 30, 31, 32]])
```

In [83]:
```
np.split(arr3,2) # Splitting into 2 parts and returns list
```

Out[83]:
```
[array([[1, 2, 3, 4],
        [5, 6, 7, 8]]),
 array([[ 9, 10, 11, 12],
        [13, 14, 15, 16]])]
```

In [84]:
```
np.split(arr3,2,axis=1)
```

Out[84]:
```
[array([[ 1,  2],
        [ 5,  6],
        [ 9, 10],
        [13, 14]]),
 array([[ 3,  4],
        [ 7,  8],
        [11, 12],
        [15, 16]])]
```

In [85]:
```
d1=np.array([4,7,1,3,9])
```

In [86]:
```
np.split(d1,[1,3,4]) #Splitting array as per our desire **ALL VALUES ARE (VALUE-1)**
```

Out[86]:
```
[array([4]), array([7, 1]), array([3]), array([9])]
```

Finding Trigonomertic Functions:

In [88]:
```
import matplotlib.pyplot as plt
```

In [89]:
```
np.sin(180)
```

Out[89]:
```
-0.8011526357338304
```

In [90]:
```
np.sin(90)
```

Out[90]:
```
0.8939966636005579
```

In [91]:
```
np.cos(180)
```

Out[91]:
```
-0.5984600690578581
```

In [92]:
```
np.tan(90)
```

Out[92]:
```
-1.995200412208242
```

In [95]:
```
x_sin=np.arange(0, 3*np.pi, 0.1)
print(x_sin)
```

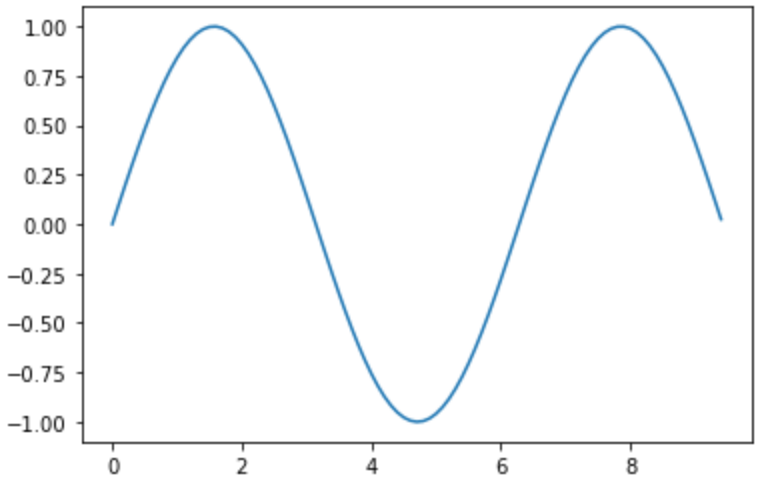
```
[0.  0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.  1.1 1.2 1.3 1.4 1.5 1.6 1.7
 1.8 1.9 2.  2.1 2.2 2.3 2.4 2.5 2.6 2.7 2.8 2.9 3.  3.1 3.2 3.3 3.4 3.5
 3.6 3.7 3.8 3.9 4.  4.1 4.2 4.3 4.4 4.5 4.6 4.7 4.8 4.9 5.  5.1 5.2 5.3
 5.4 5.5 5.6 5.7 5.8 5.9 6.  6.1 6.2 6.3 6.4 6.5 6.6 6.7 6.8 6.9 7.  7.1
 7.2 7.3 7.4 7.5 7.6 7.7 7.8 7.9 8.  8.1 8.2 8.3 8.4 8.5 8.6 8.7 8.8 8.9
 9.  9.1 9.2 9.3 9.4]
```

In [96]:
```
y_sin=np.sin(x_sin)
print(y_sin)
```

```
[ 0.          0.09983342  0.19866933  0.29552021  0.38941834  0.47942554
  0.56464247  0.64421769  0.71735609  0.78332691  0.84147098  0.89120736
  0.93203909  0.96355819  0.98544973  0.99749499  0.9995736   0.99166481
  0.97384763  0.94630009  0.90929743  0.86320937  0.8084964   0.74570521
  0.67546318  0.59847214  0.51550137  0.42737988  0.33498815  0.23924933
  0.14112001  0.04158066 -0.05837414 -0.15774569 -0.2555411  -0.35078323
 -0.44252044 -0.52983614 -0.61185789 -0.68776616 -0.7568025  -0.81827711
 -0.87157577 -0.91616594 -0.95160207 -0.97753012 -0.993691   -0.99992326
 -0.99616461 -0.98245261 -0.95892427 -0.92581468 -0.88345466 -0.83226744
 -0.77276449 -0.70554033 -0.63126664 -0.55068554 -0.46460218 -0.37387666
 -0.2794155  -0.1821625  -0.0830894   0.0168139   0.1165492   0.21511999
  0.31154136  0.40484992  0.49411335  0.57843976  0.6569866   0.72896904
  0.79366786  0.85043662  0.8987081   0.93799998  0.96791967  0.98816823
  0.99854335  0.99894134  0.98935825  0.96988981  0.94073056  0.90217183
  0.85459891  0.79848711  0.7343971   0.66296923  0.58491719  0.50102086
  0.41211849  0.31909836  0.22288991  0.12445442  0.02477543]
```

In [97]:
```
plt.plot(x_sin,y_sin) # to plot the graph (x-y)
plt.show # show function helps to visualize
```

```
<function matplotlib.pyplot.show(close=None, block=None)>
```

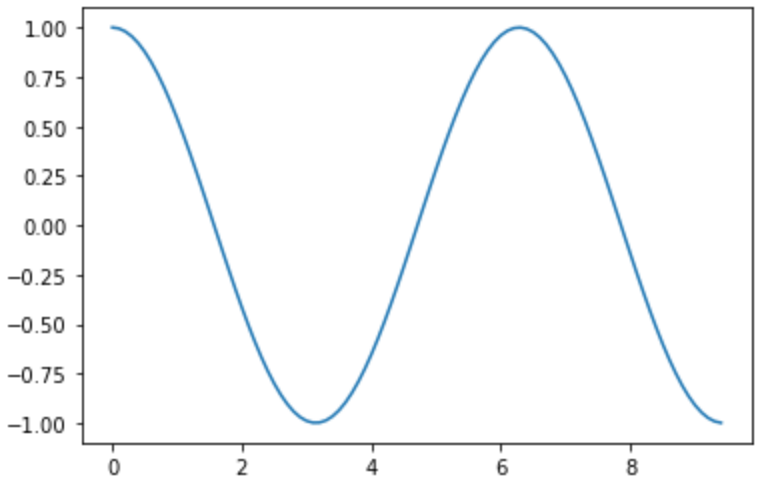


In [99]:
```
y_cos=np.cos(x_sin)
print(y_cos)
```

```
[ 1.          0.99500417  0.98006658  0.95533649  0.92106099  0.87758256
  0.82533561  0.76484219  0.69670671  0.62160997  0.54030231  0.45359612
  0.36235775  0.26749883  0.16996714  0.0707372  -0.02919952 -0.12884449
 -0.22720209 -0.32328957 -0.41614684 -0.5048461  -0.58850112 -0.66627602
 -0.73739372 -0.80114362 -0.85688875 -0.90407214 -0.94222234 -0.97095817
 -0.9899925  -0.99913515 -0.99829478 -0.98747977 -0.96679819 -0.93645669
 -0.89675842 -0.84810003 -0.79096771 -0.7259323  -0.65364362 -0.57482395
 -0.49026082 -0.40079917 -0.30733287 -0.2107958  -0.11215253 -0.01238866
  0.08749898  0.18651237  0.28366219  0.37797774  0.46851667  0.55437434
  0.63469288  0.70866977  0.77556588  0.83471278  0.88551952  0.92747843
  0.96017029  0.98326844  0.9965421   0.99985864  0.99318492  0.97658763
  0.95023259  0.91438315  0.86939749  0.8157251   0.75390225  0.68454667
  0.60835131  0.52607752  0.43854733  0.34663532  0.25125984  0.15337386
  0.05395542 -0.04600213 -0.14550003 -0.24354415 -0.33915486 -0.43137684
 -0.51928865 -0.6020119  -0.67872005 -0.74864665 -0.81109301 -0.86543521
 -0.91113026 -0.9477216  -0.97484362 -0.99222533 -0.99969304]
```

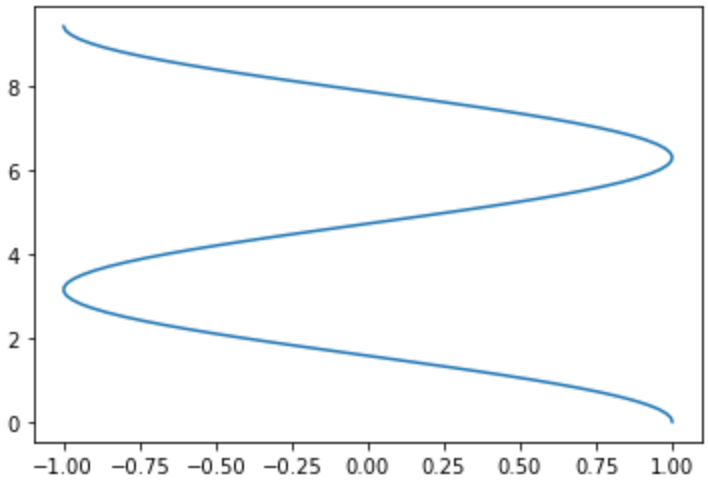
In [100…
```
plt.plot(x_sin,y_cos) # to plot the graph (x-y)
plt.show # show function helps to visualize
```

Out[100…
```
<function matplotlib.pyplot.show(close=None, block=None)>
```



In [101…
```
plt.plot(y_cos,x_sin) # to plot the graph (y-x)
plt.show # show function helps to visualize
```

Out[101…
```
<function matplotlib.pyplot.show(close=None, block=None)>
```



In [102…

```
y_tan=np.tan(x_sin)
print(y_tan)
```

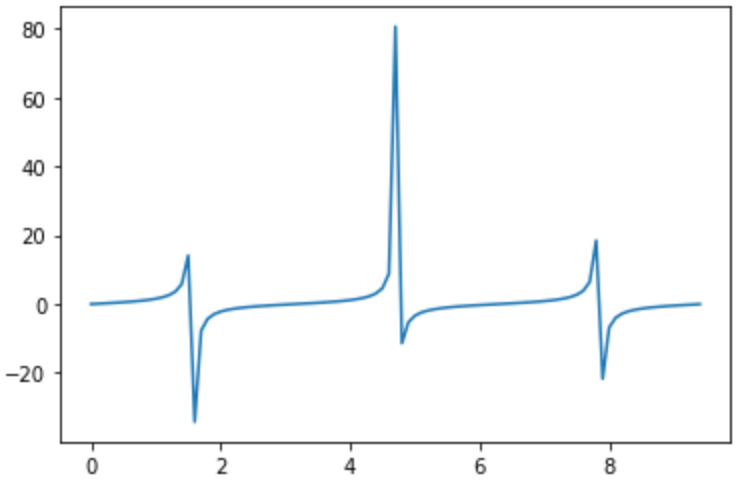
```
[ 0.00000000e+00  1.00334672e-01  2.02710036e-01  3.09336250e-01
  4.22793219e-01  5.46302490e-01  6.84136808e-01  8.42288380e-01
  1.02963856e+00  1.26015822e+00  1.55740772e+00  1.96475966e+00
  2.57215162e+00  3.60210245e+00  5.79788372e+00  1.41014199e+01
 -3.42325327e+01 -7.69660214e+00 -4.28626167e+00 -2.92709751e+00
 -2.18503986e+00 -1.70984654e+00 -1.37382306e+00 -1.11921364e+00
 -9.16014290e-01 -7.47022297e-01 -6.01596613e-01 -4.72727629e-01
 -3.55529832e-01 -2.46405394e-01 -1.42546543e-01 -4.16166546e-02
  5.84738545e-02  1.59745748e-01  2.64316901e-01  3.74585640e-01
  4.93466730e-01  6.24733075e-01  7.73556091e-01  9.47424650e-01
  1.15782128e+00  1.42352648e+00  1.77777977e+00  2.28584788e+00
  3.09632378e+00  4.63733205e+00  8.86017490e+00  8.07127630e+01
 -1.13848707e+01 -5.26749307e+00 -3.38051501e+00 -2.44938942e+00
 -1.88564188e+00 -1.50127340e+00 -1.21754082e+00 -9.95584052e-01
 -8.13943284e-01 -6.59730572e-01 -5.24666222e-01 -4.03110900e-01
 -2.91006191e-01 -1.85262231e-01 -8.33777149e-02  1.68162777e-02
  1.17348947e-01  2.20277200e-01  3.27858007e-01  4.42757417e-01
  5.68339979e-01  7.09111151e-01  8.71447983e-01  1.06489313e+00
  1.30462094e+00  1.61656142e+00  2.04928417e+00  2.70601387e+00
  3.85226569e+00  6.44287247e+00  1.85068216e+01 -2.17151127e+01
 -6.79971146e+00 -3.98239825e+00 -2.77374930e+00 -2.09137751e+00
 -1.64571073e+00 -1.32636433e+00 -1.08203242e+00 -8.85556937e-01
 -7.21146876e-01 -5.78923588e-01 -4.52315659e-01 -3.36700526e-01
 -2.28641712e-01 -1.25429598e-01 -2.47830328e-02]
```

In [103…

```
plt.plot(x_sin,y_tan) # to plot the graph (x-y)
plt.show # show function helps to visualize
```

Out[103…

```
<function matplotlib.pyplot.show(close=None, block=None)>
```



Random Sampling: To generate data

In [104…
```
import random
```

In [105…
```
np.random.random(1)
```

Out[105…
```
array([0.65920157])
```

In [107…
```
np.random.random((3,3))
```

Out[107…
```
array([[0.83962408, 0.92296159, 0.55802282],
       [0.19465548, 0.2300537 , 0.69830695],
       [0.01701227, 0.54915568, 0.85469192]])
```

In [110…
```
np.random.randint(1,4)
```

Out[110…
```
2
```

In [111…
```
np.random.randint(1,4,(4,4)) #(4,4) is in tuple to create 2D array cux here we don't have
```

Out[111…
```
array([[2, 2, 1, 1],
       [2, 3, 2, 1],
       [2, 3, 1, 1],
       [1, 2, 1, 1]])
```

In [112…
```
np.random.randint(1,4,(2,4,4)) #(2,4,4) is in tuple to create 3D array cux here we don't
```

Out[112…
```
array([[[1, 1, 2, 2],
        [3, 3, 2, 3],
        [2, 1, 3, 2],
        [2, 2, 1, 3]],

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

In [114…
```
np.random.seed(10) #seed function is used to create similar array everytime we want
np.random.randint(1,4,(2,4,4))
```

Out[114…
```
array([[[2, 2, 1, 1],
        [2, 1, 2, 2],
        [1, 2, 2, 3],
        [1, 2, 1, 3]],

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

In [115…
```
np.random.seed(10) #seed function is used to create similar array everytime we want so it
# Seed Fuction can take values upto (2**32-1=4294967295)
np.random.randint(1,4,(2,4,4))
```

Out[115…
```
array([[[2, 2, 1, 1],
        [2, 1, 2, 2],
        [1, 2, 2, 3],
        [1, 2, 1, 3]],

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

In [116…
```
np.random.rand(3) # how much elements we want
```

Out[116…
```
array([0.13145815, 0.41366737, 0.77872881])
```

In [117…
```
np.random.rand(3,3) # how much elements we want
```

Out[117…
```
array([[0.58390137, 0.18263144, 0.82608225],
       [0.10540183, 0.28357668, 0.06556327],
       [0.05644419, 0.76545582, 0.01178803]])
```

In [118…
```
np.random.randn(3,3) # how much elements we want given negatives values also
```

Out[118…
```
array([[-1.58494101,  1.05535316, -1.92657911],
       [ 0.69858388, -0.74620143, -0.15662666],
       [-0.19363594,  1.13912535,  0.36221796]])
```

In [119…
```
l=[1,2,3,4,5]
np.random.choice(l)
```

Out[119…
```
1
```

In [120…
```
l=[1,2,3,4,5]
np.random.choice(l)
```

Out[120…
```
1
```

In [121…
```
for ch in range(10):
    l=[1,2,3,4,5]
    print(np.random.choice(l))
```

```
5
5
1
```

```
3
5
3
1
1
3
```

In [122…
```
np.random.permutation(l)
```

Out[122…
```
array([2, 3, 5, 1, 4])
```

In [123…
```
np.random.permutation(l)
```

Out[123…
```
array([5, 2, 4, 3, 1])
```

String Operations, Comparision and Information

In [141…
```
s1='Indian Air Force'
s2=' Indian Navy'
```

In [143…
```
np.char.add(s1,s2) #Numpy has char module which has add method for string datas
```

Out[143…
```
array('Indian Air Force Indian Navy', dtype='<U28')
```

In [142…
```
s1+s2
```

Out[142…
```
'Indian Air Force Indian Navy'
```

In [146…
```
np.char.lower(s1)
```

Out[146…
```
array('indian air force', dtype='<U16')
```

In [147…
```
np.char.upper(s2)
```

Out[147…
```
array(' INDIAN NAVY', dtype='<U12')
```

In [148…
```
np.char.center(s1,60) #Total space we want
```

Out[148…
```
array('                      Indian Air Force                      ',
      dtype='<U60')
```

In [149…
```
np.char.center(s1,60,fillchar='*') #We want to fill white spaces with desired
```

Out[149…
```
array('**********Indian Air Force**********',
      dtype='<U60')
```

In [150…
```
np.char.split(s1)
```

```
array(list(['Indian', 'Air', 'Force']), dtype=object)
```

In [152…
```
np.char.splitlines('Hello\nShubhlife') # To split chars attched with lines
```

Out[152…
```
array(list(['Hello', 'Shubhlife']), dtype=object)
```

In [155…
```
np.char.join([':','/'],[s1,s2])
```

Out[155…
```
array(['I:n:d:i:a:n: :A:i:r: :F:o:r:c:e', ' /I/n/d/i/a/n/ /N/a/v/y'],
      dtype='<U31')
```

In [157…
```
np.char.replace(s2,'Navy','ISRO')
```

Out[157…
```
array(' Indian ISRO', dtype='<U12')
```

In [158…
```
np.char.equal(s1,s2)
```

Out[158…
```
array(False)
```

In [161…
```
np.char.count(s1,'a')
```

Out[161…
```
array(1)
```

In [162…
```
np.char.find(s1,'a')
```

Out[162…
```
array(4)
```

In [ ]:
```

```