In [ ]:
```
import math
math.sqrt(25)
```

In [ ]:
```
# How to get input from user
num = input("Enter a number ")
print('Entered ',num, ' of type', type(num))
#By default the input value is considered as a string , we can convert it int
```

In [ ]:
```
string = input("What is your name?")
print('Entered ',string, ' of type', type(num))
num1 = int(input("Enter a number"))
print('Entered ',num1, ' of type', type(num1))
```

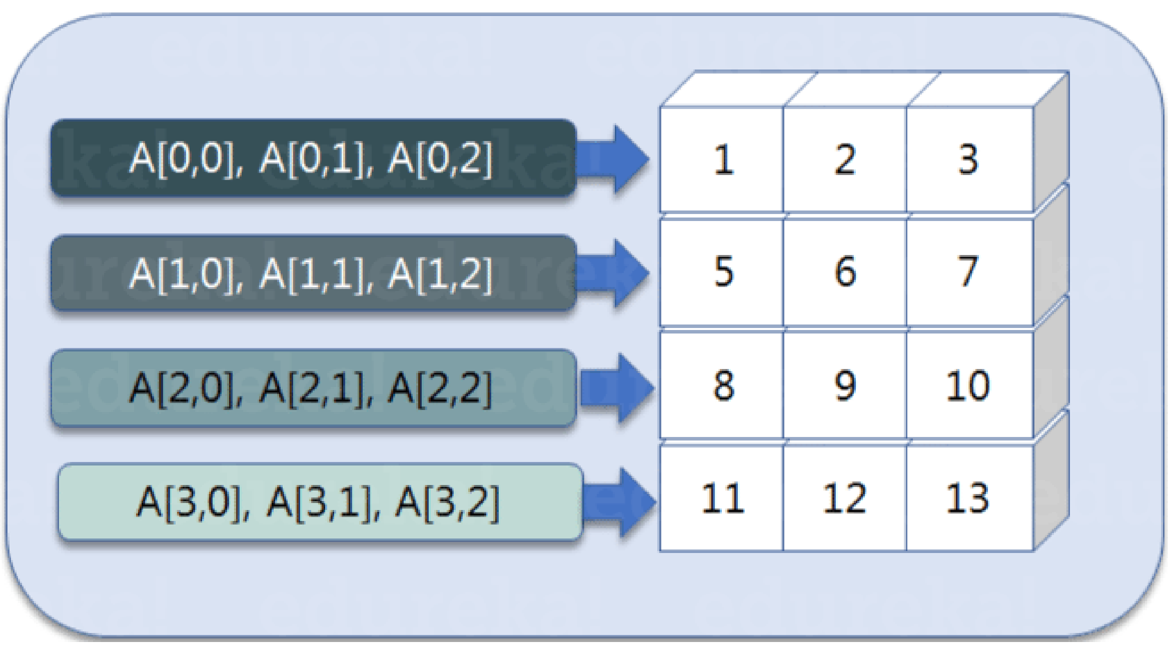
Python NumPy

#To install do pip install numpy

What is a Python NumPy?

NumPy is a Python package which stands for 'Numerical Python'. It is the core library for scientific computing, which contains a powerful n-dimensional array object, provide tools for integrating C, C++ etc. It is also useful in linear algebra, random number capability etc. NumPy array can also be used as an efficient multi-dimensional container for generic data.

NumPy Array: Numpy array is a powerful N-dimensional array object which is in the form of rows and columns. We can initialize numpy arrays from nested Python lists and access it elements.



Here, I have different elements that are stored in their respective memory locations. It is said to be two dimensional because it has rows as well as columns. In the above image, we have 3 columns and 4 rows available.

Single & Multi dimensional Numpy Array:

In [1]:
```
import numpy as np
a=np.array([1,2,3])
print(a)
```

```
[1 2 3]
```

In [2]:
```
#Multi-dimensional Array:
a=np.array([(1,2,3),(4,5,6)])
print(a)
```

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

Python NumPy Array v/s List

We use python numpy array instead of a list because of the below three reasons:

Less Memory

Fast

Convenient

The very first reason to choose python numpy array is that it occupies less memory as compared to list. Then, it is pretty fast in terms of execution and at the same time it is very convenient to work with numpy. So these are the major advantages that python numpy array has over list.

In [ ]:
```
sys.getsizeof(S)
```

In [4]:

```
import numpy as np

import time
import sys
S= range(1000)
# print(list(S))
print(S)
print(sys.getsizeof(S))
print(sys.getsizeof(S)*len(S))

D= np.arange(1000)
print(D.size*D.itemsize)
# D
```

```
range(0, 1000)
48
48000
4000
```

The above output shows that the memory allocated by list (denoted by S) is 48000 whereas the memory allocated by the numpy array is just 4000. From this, you can conclude that there is a major difference between the two and this makes python numpy array as the preferred choice over list.

In [ ]:

```

```

In [6]:

```
#python numpy array is faster and more convenient when compared to list
# import time
# import sys

SIZE = 1000000

L1= range(SIZE)
L2= range(SIZE)
A1= np.arange(SIZE)
A2=np.arange(SIZE)

start= time.time()
# print(start)
result=[x+y for x,y in zip(L1,L2)]
print((time.time()-start)*1000)

start=time.time()
result= A1+A2
print((time.time()-start)*1000)
```

```
111.70077323913574
18.97740364074707
```

In [7]:

```
result
```

Out[7]:

```
array([      0,       2,       4, ..., 1999994, 1999996, 1999998])
```

In [8]:
```
# list(
zip(L1,L2)
```

Out[8]:
```
<zip at 0x28448946608>
```

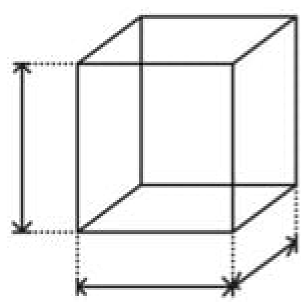
In the above code, we have defined two lists and two numpy arrays. Then, we have compared the time taken in order to find the sum of lists and sum of numpy arrays both. If you see the output of the above program, there is a significant change in the two values. List took 380ms whereas the numpy array took almost 49ms. Hence, numpy array is faster than list. Now, if you noticed we had run a 'for' loop for a list which returns the concatenation of both the lists whereas for numpy arrays, we have just added the two array by simply printing A1+A2. That's why working with numpy is much easier and convenient when compared to the lists.

Python NumPy Operations

In [9]:
```
#ndim:
# import numpy as np
a = np.array([(1,2,3),(4,5,6),(8,9,10)])
print(a.ndim) #Since the output is 2, it is a two-dimensional array (multi di
a.size
```

```
2
```

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



itemsize: You can calculate the byte size of each element. In the below code, I have defined a single dimensional array and with the help of 'itemsize' function, we can find the size of each element.

In [10]:
```
# import numpy as np
a = np.array([(1,2,3)])
print(a.itemsize)
```

```
4
```

dtype: You can find the data type of the elements that are stored in an array. So, if you want to know the data type of a particular element, you can use 'dtype' function which will print the datatype along with the size. In the below code, I have defined an array where I have used the same function.

In [11]:
```
# import numpy as np
a = np.array([(1,1,1)])
print(a.dtype)
```

```
int32
```

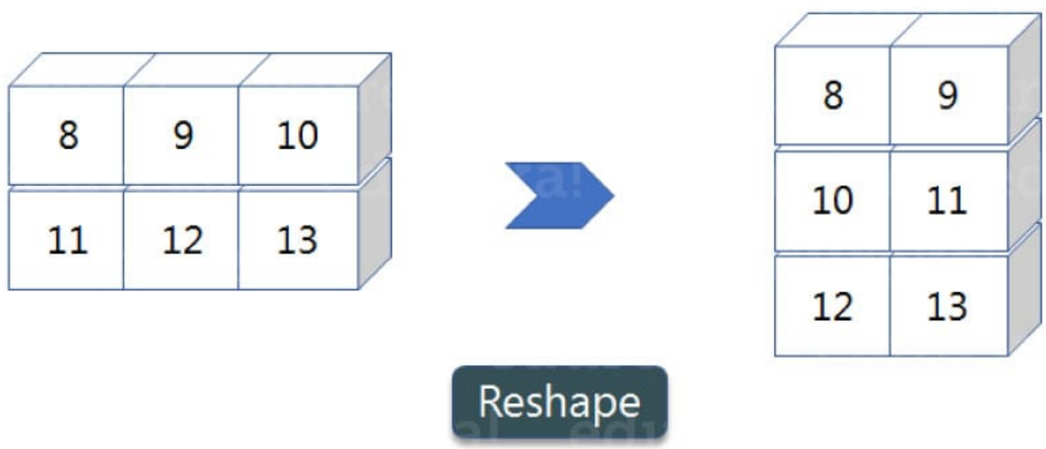
In [12]:
```
# Calculating size and shape of an array
# import numpy as np

a = np.array([(1,2,3),(4,5,6)])
print(a.size)
print(a.shape)
```

```
6
(2, 3)
```

reshape:

Reshape is when you change the number of rows and columns which gives a new view to an object.



As you can see in the above image, we have 3 columns and 2 rows which has converted into 2 columns and 3 rows

In [13]:
```
import numpy as np
a = np.array([(8,9,10),(11,12,13)])
print('Old -->',a)
a=a.reshape(3,2)
print('New-->',a)
a.dtype
```

```
Old --> [[ 8  9 10]
 [11 12 13]]
New--> [[ 8  9]
 [10 11]
 [12 13]]
```

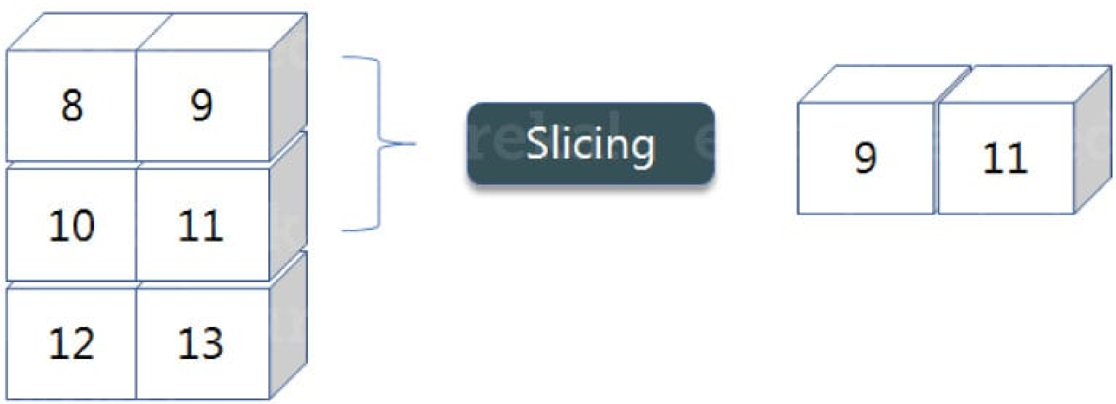
Out[13]:
```
dtype('int32')
```

In [14]:
```
test = np.array([(1,2,3,4,5,6),(1,2,3,4,5,6),(1,2,3,4,5,6)])
test.reshape(9,2)
```

Out[14]:
```
array([[1, 2],
       [3, 4],
       [5, 6],
       [1, 2],
       [3, 4],
       [5, 6],
       [1, 2],
       [3, 4],
       [5, 6]])
```

slicing:

Slicing is basically extracting particular set of elements from an array. This slicing operation is pretty much similar to the one which is there in the list as well.



In [15]:
```
#We have an array and we need a particular element (say 3) out of a given arr
# import numpy as np
a=np.array([(1,2,3,4),(3,4,5,6)])
print(a)
print(a[0,2])
#Here, the array(1,2,3,4) is your index 0 and (3,4,5,6) is index 1 of the pyt
#Therefore, we have printed the second element from the zeroth index.
```

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

In [16]:
```
#Let's say we need the 2nd element from the zeroth and first index of the arr
import numpy as np
a=np.array([(1,2,3,4),(3,4,5,6)])
print(a)
print(a[0:,2]) # Here colon represents all the rows, including zero.
               #Now to get the 2nd element, we'll call index 2 from both of t
```

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

In [17]:
```
import numpy as np
a=np.array([(8,9,11),(10,11,12),(12,13,13),(3,5,6)])
print(a)
print(a[0:3,1])
#As you can see in the above code, only 9 and 11 gets printed. Now when I hav
#Therefore, only 9 and 11 gets printed else you will get all the elements i.e
```

```
[[ 8  9 11]
 [10 11 12]
 [12 13 13]
 [ 3  5  6]]
[ 9 11 13]
```

linspace:

This is another operation in python numpy which returns evenly spaced numbers over a specified interval.

In [18]:
```
list(range(0,10,2))
```

Out[18]: `[0, 2, 4, 6, 8]`

In [19]:
```
import numpy as np
a=np.linspace(1,100,10)
print(a) #it has printed 10 values between 1 to 3.
```

```
[  1.  12.  23.  34.  45.  56.  67.  78.  89. 100.]
```

Min, max, mean, sum ,Square Root, Standard Deviationetc

In [20]:
```
import numpy as np

a= np.array([19,23,56,10,19,76,84,90,12])
print(a.min())
print(a.max())
print(a.sum())
print(a.mean())
print(np.sqrt(a))
print(np.std(a))
```

```
10
90
389
43.22222222222222
[4.35889894 4.79583152 7.48331477 3.16227766 4.35889894 8.71779789
 9.16515139 9.48683298 3.46410162]
31.179686327899013
```

In [21]:
```
a=np.array([(8,9),(10,11),(12,13)])
print(a.min())
print(a.max())
print(np.sum(a))
print(a.mean())
print(np.sqrt(a))
print(np.std(a))
```

```
8
13
63
10.5
[[2.82842712 3.        ]
 [3.16227766 3.31662479]
 [3.46410162 3.60555128]]
1.707825127659933
```

Calculating mean, median with numpy inbuilt functions

In [22]:
```
import numpy as np

# 1D array
arr = [20, 2, 7, 1, 34]

print("arr : ", arr)
print("median of arr : ", np.median(arr))
```

```
arr :  [20, 2, 7, 1, 34]
median of arr :  7.0
```

In [23]:
```
import numpy as np

# 2D array
arr = [[14, 17, 12, 33, 44],
       [15, 6, 27, 8, 19],
       [23, 2, 54, 1, 4 ]]

# median of the flattened array
print("\nmedian of arr, axis = None : ", np.median(arr))
print("\nmean of arr, axis = None : ", np.mean(arr))

# median along the axis = 0
print("\nmedian of arr, axis = 0 : ", np.median(arr, axis = 0))
print("\nmean of arr, axis = 0 : ", np.mean(arr, axis = 0))

# median along the axis = 1
print("\nmedian of arr, axis = 1 : ", np.median(arr, axis = 1))
print("\nmean of arr, axis = 1 : ", np.mean(arr, axis = 1))

out_arr = np.arange(3)
print("\nout_arr : ", out_arr)
print("median of arr, axis = 1 : ",
      np.median(arr, axis = 1, out = out_arr))
```

```
median of arr, axis = None :  15.0

mean of arr, axis = None :  18.6

median of arr, axis = 0 :  [15.  6. 27.  8. 19.]

mean of arr, axis = 0 :  [17.33333333  8.33333333 31.         14.         2
2.33333333]

median of arr, axis = 1 :  [17. 15.  4.]

mean of arr, axis = 1 :  [24.  15.  16.8]

out_arr :  [0 1 2]
median of arr, axis = 1 :  [17 15  4]
```

In [24]:
```
out_arr
```

Out[24]:
```
array([17, 15,  4])
```

In [25]:
```
# a = np.array([1,2,3])
```

In [26]:
```
# print(arr)
# np.sum(arr,axis=0)
```

Addition Operation

In [27]:

```
#You can perform more operations on numpy array i.e addition, subtraction,mul
import numpy as np
x= np.array([(1,2,3),(3,4,5)])
y= np.array([(1,2,3),(3,4,5)])
print(x+y)
print(x-y)
print(x*y)
print(x/y)
```

```
[[ 2  4  6]
 [ 6  8 10]]
[[0 0 0]
 [0 0 0]]
[[ 1  4  9]
 [ 9 16 25]]
[[1. 1. 1.]
 [1. 1. 1.]]
```

Vertical & Horizontal Stacking

if you want to concatenate two arrays and not just add them, you can perform it using two ways – vertical stacking and horizontal stacking.

In [28]:

```
import numpy as np
x= np.array([(1,2,3),(3,4,5)])
y= np.array([(1,2,3),(3,4,65)])
print(np.vstack((x,y)))
print(np.hstack((x,y)))
```

```
[[ 1  2  3]
 [ 3  4  5]
 [ 1  2  3]
 [ 3  4 65]]
[[ 1  2  3  1  2  3]
 [ 3  4  5  3  4 65]]
```

ravel

There is one more operation where you can convert one numpy array into a single column i.e ravel.

In [29]:

```
import numpy as np
import math
x= np.array([(1,2,3),(3,4,5)])
print(x)
print(x.ravel())
print(x.flatten())
```

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

In [30]:
```
y = x.ravel()
y[0]=11
z = x.flatten()
z[0] =22
```

In [31]:
```
print(z)
print(y)
x
```

```
[22  2  3  3  4  5]
[11  2  3  3  4  5]
```

Out[31]:
```
array([[11,  2,  3],
       [ 3,  4,  5]])
```

In [32]:
```
# All constants
np.full((3,3), math.pi)
```

Out[32]:
```
array([[3.14159265, 3.14159265, 3.14159265],
       [3.14159265, 3.14159265, 3.14159265],
       [3.14159265, 3.14159265, 3.14159265]])
```

In [33]:
```
# One more example with all constants
np.full((3,2),4,dtype=float)
```

Out[33]:
```
array([[4., 4.],
       [4., 4.],
       [4., 4.]])
```

In [34]:
```
# Identity Matrix
np.eye(5)
```

Out[34]:
```
array([[1., 0., 0., 0., 0.],
       [0., 1., 0., 0., 0.],
       [0., 0., 1., 0., 0.],
       [0., 0., 0., 1., 0.],
       [0., 0., 0., 0., 1.]])
```

In [35]:
```
# Random numbers from [0,1]
np.random.random((2,2))
```

Out[35]:
```
array([[0.6770599 , 0.96073638],
       [0.12115127, 0.8886565 ]])
```

As a side note , single random number from [0,1) can be obtained like this

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

Out[36]:
```
array([0.36586098, 0.68389728, 0.30556904, 0.90182525])
```

To obtain a number in the interval [a,b), you can simply multiply above with (b-a) and then add a.

In [37]:
```
print(np.zeros((2,2)))
np.ones((2,2))
```

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

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

In [38]:
```
# another example

a = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
a
```

Out[38]:
```
array([[ 1,  2,  3],
       [ 4,  5,  6],
       [ 7,  8,  9],
       [10, 11, 12]])
```

In [39]:
```
b = np.array([0, 2, 0, 1])
print(b)
c=np.arange(4)
c
```

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

Out[39]:
```
array([0, 1, 2, 3])
```

In [40]:
```
#0201 -- b
#0123 --c
# 0,0\
# 1,2
# 2,0
# 3,1
```

In [41]:
```
a[c,b]
```

Out[41]:
```
array([ 1,  6,  7, 11])
```

Using index you can access elements as well as modify them

In [42]:
```
a
```

Out[42]:
```
array([[ 1,  2,  3],
       [ 4,  5,  6],
       [ 7,  8,  9],
       [10, 11, 12]])
```

In [43]:
```
a[c, b] += 10
a
```

Out[43]:
```
array([[11,  2,  3],
       [ 4,  5, 16],
       [17,  8,  9],
       [10, 21, 12]])
```

Conditional Access of arrays

if a here was a single element , wirintg a>2 wil gegenrate True or False depeneding on whetrher that particular consition was true.

Now when a is a numpy array, that comparison will be done for each element and result will be an array of shape same as a ; containing True/False for each element.

In [44]:
```
a = np.array([[1,2], [3, 4], [5, 6]])
a
```

Out[44]:
```
array([[1, 2],
       [3, 4],
       [5, 6]])
```

In [45]:
```
c=a > 2
print(c)
```

```
[[False False]
 [ True  True]
 [ True  True]]
```

We can use , these comparison expressions directly for access. Result is only those elements for which the expression evaluates to True

In [46]:
```
print(a[c])
print(a[c].shape)
```

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

notice that the result is a 1D array.

Lets see if this works with writing mulitple conditions as well. In that process we'll also see that we dont have to store results in one variable and then pass for subsetting. We can instead, write the conditional expression directly for subsetting.

In [47]:
```
a[(a>2) | (a<5)]
```

Out[47]:
```
array([1, 2, 3, 4, 5, 6])
```

In [48]:
```
a[(a>2) & (a<5)]
```
Out[48]:
```
array([3, 4])
```

Array Operations

We'll see that you can use ; both normal symbols as well as numpy functions for array operations. Lets look at these operations with examples

In [49]:
```
x = np.array([[1,2],[3,4]])
y = np.array([[5,6],[7,8]])
```
In [50]:
```
x
```
Out[50]:
```
array([[1, 2],
       [3, 4]])
```
In [51]:
```
y
```
Out[51]:
```
array([[5, 6],
       [7, 8]])
```
In [52]:
```
x+y
```
Out[52]:
```
array([[ 6,  8],
       [10, 12]])
```
In [53]:
```
np.add(x,y)
```
Out[53]:
```
array([[ 6,  8],
       [10, 12]])
```
In [54]:
```
print(x-y)
```
```
[[-4 -4]
 [-4 -4]]
```
In [55]:
```
np.subtract(x,y)
```
Out[55]:
```
array([[-4, -4],
       [-4, -4]])
```

In [56]:
```
# element wise multiplication , not matrix multiplication
print(x)
print("~~~~~")

print(y)
print("~~~~~")
print(x * y)
```

```
[[1 2]
 [3 4]]
~~~~~
[[5 6]
 [7 8]]
~~~~~
[[ 5 12]
 [21 32]]
```

In [57]:
```
np.multiply(x, y)
```

Out[57]:
```
array([[ 5, 12],
       [21, 32]])
```

In [58]:
```
print(x/y)
```

```
[[0.2        0.33333333]
 [0.42857143 0.5       ]]
```

In [59]:
```
np.divide(x,y)
```

Out[59]:
```
array([[0.2       , 0.33333333],
       [0.42857143, 0.5       ]])
```

In general you'll find that , mathematical functions from numpy [being referred as np here] when applied on array, give back result as an array where that function has been applied on individual elements. These function from package math on the other hand give error when applied to arrays. They only work for scalars.

In [60]:
```
x
```

Out[60]:
```
array([[1, 2],
       [3, 4]])
```

In [61]:
```
np.sqrt(x)
```

Out[61]:
```
array([[1.        , 1.41421356],
       [1.73205081, 2.        ]])
```

In [62]:
```
math.sqrt(x)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-62-b33d9061ea8b> in <module>
----> 1 math.sqrt(x)

TypeError: only size-1 arrays can be converted to Python scalars
```

In [63]:
```
v = np.array([9,10])
v
```

Out[63]:
```
array([ 9, 10])
```

In [64]:
```
w = np.array([11, 12])
w
```

Out[64]:
```
array([11, 12])
```

In [65]:
```
# Matrix multiplication
v.dot(w)
```

Out[65]:
```
219
```

You can see that result is not what you'd expect from matrix multiplication. This happens because a single dimensional array is not a matrix.

In [66]:
```
# 1*2 -- 2*1
```

In [67]:
```
v=v.reshape((1,2))
w=w.reshape((1,2))
```

Now if you simply try to do v.dot(w) or np.dot(v,w) [both are same] , you will get and error because you can multiple a mtrix of shape 2X1 with a matrix of 2X1 .

In [68]:
```
np.dot(v,w)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-68-efb51945670c> in <module>
----> 1 np.dot(v,w)

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

In [69]:
```
print('matrix v : ',v)
print('matrix v Transpose:',v.T)
print('matrix w:',w)
print('matrix w Transpose:',w.T)
print('~~~~~~~~~')
print(np.dot(v,w.T))
print('~~~~~~~~~')
print(np.dot(v.T,w))
```

```
matrix v :  [[ 9 10]]
matrix v Transpose: [[ 9]
 [10]]
matrix w: [[11 12]]
matrix w Transpose: [[11]
 [12]]
~~~~~~~~~
[[219]]
~~~~~~~~~
[[ 99 108]
 [110 120]]
```

If you leave v to be a single dimensional array . you will simply get an element wise multiplication. Here is an example

In [70]:
```
print(x)
print("~~~")
print(y)
x.dot(y)
```

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

Out[70]:
```
array([[19, 22],
       [43, 50]])
```

other functions

In [71]:
```
x = np.array([[1,2],[3,4]])
x
```

Out[71]:
```
array([[1, 2],
       [3, 4]])
```

In [72]:
```
np.sum(x)
```

Out[72]:
```
10
```

Using axis option in the function sum , you can some across both the dimension of array separately as well

In [73]:
```
np.sum(x, axis=0)
```

Out[73]:
```
array([4, 6])
```

In [74]:
```
np.sum(x, axis=1)
```

Out[74]:
```
array([3, 7])
```

So far we have seen that, when we do operations between two arrays; operation happens between corresponding elements of the arrays. Many at times , shape of arrays will not match and correspondence between elements will not be complete. In such case , elements of the smaller array are recycled to makeup for the correspondence.

In [75]:
```
x = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
v = np.array([1, 0, 1])
v
```

Out[75]:
```
array([1, 0, 1])
```

here v is a smaller array than x, lets see what happens when we do operation between x and v. But before that , we are going to replicate v to make up for the correpondence ourselves and see the result

In [76]:
```
vv = np.tile(v, (4, 1))  # Stack 4 copies of v on top of each other
vv
```

Out[76]:
```
array([[1, 0, 1],
       [1, 0, 1],
       [1, 0, 1],
       [1, 0, 1]])
```

In [77]:
```
print(np.hstack((v,v)))
np.vstack((v,v))
```

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

Out[77]:
```
array([[1, 0, 1],
       [1, 0, 1]])
```

In [78]:
```
print(x)
print("~~~~~")
print(vv)
x + vv
```

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

Out[78]:
```
array([[ 2,  2,  4],
       [ 5,  5,  7],
       [ 8,  8, 10],
       [11, 11, 13]])
```

Python Numpy Special Functions

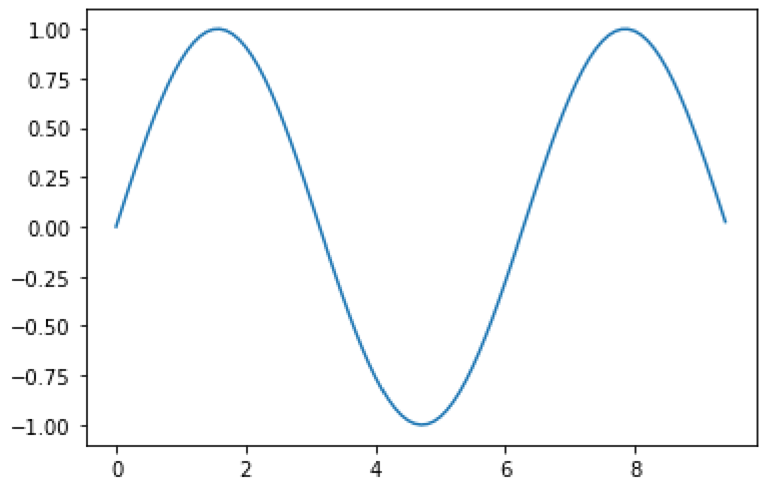
In [79]:
```
np.sin(np.array((0., 30., 45., 60., 90.)) * np.pi / 180. )
```

Out[79]:
```
array([0.        , 0.5       , 0.70710678, 0.8660254 , 1.        ])
```

In [80]:

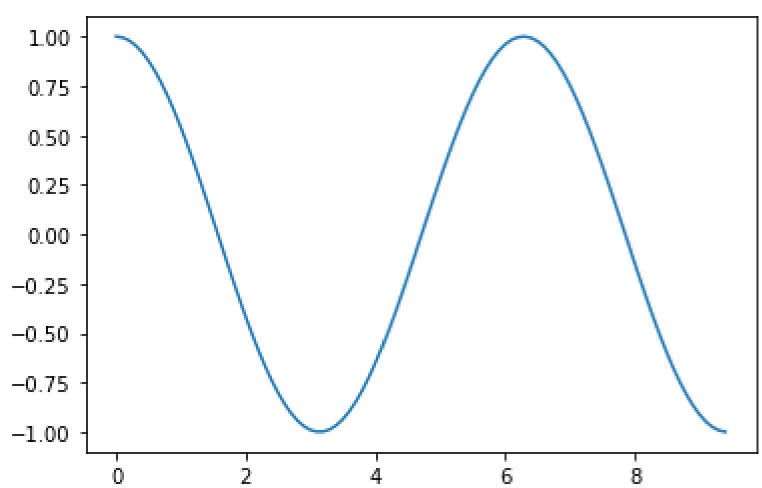
```
#There are various special functions available in numpy such as sine, cosine,
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
x= np.arange(0,3*np.pi,0.1)
print(x)
y=np.sin(x)
plt.plot(x,y)
plt.show()
```

```
[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 [81]:

```
import numpy as np
import matplotlib.pyplot as plt
x= np.arange(0,3*np.pi,0.1)
y=np.cos(x)
plt.plot(x,y)
plt.show()
```



In [82]:
```
#Exp
a= np.array([1,2,3])
print(np.exp(a))
```

```
[ 2.71828183  7.3890561  20.08553692]
```

In [83]:
```
np.arange(10)
```

Out[83]:
```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

In [84]:
```
#log
import numpy as np
import matplotlib.pyplot as plt
a= np.array([1,2,3])
print(np.log(a))
```

```
[0.         0.69314718 1.09861229]
```

Creating Identity matrix,zero matrix , matrix multiplication using numpy

In [85]:
```
#Identity matrix
import numpy as np

# 2x2 matrix with 1's on main diagnol
b = np.identity(2, dtype = float)
print("Matrix b : \n", b)


a = np.identity(4)
print("\nMatrix a : \n", a)
```

```
Matrix b :
 [[1. 0.]
 [0. 1.]]

Matrix a :
 [[1. 0. 0. 0.]
 [0. 1. 0. 0.]
 [0. 0. 1. 0.]
 [0. 0. 0. 1.]]
```

In [86]:
```
#Zero matrix
import numpy as np

# 2x2 matrix with 1's on main diagnol
b = np.zeros((2,2), dtype = float)
print("Matrix b : \n", b)


a = np.zeros((4,4))
print("\nMatrix a : \n", a)
```

```
Matrix b :
 [[0. 0.]
 [0. 0.]]

Matrix a :
 [[0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]]
```

In [87]:
```
#Matrix multiplication
a = np.array([[1, 0],[0, 1]])
b = np.array([[4, 1],[2, 2]])
np.matmul(a, b)
```

Out[87]:
```
array([[4, 1],
       [2, 2]])
```

In [88]:
```
#Matrix transpose
x = np.arange(4).reshape((2,2))
x
```

Out[88]:
```
array([[0, 1],
       [2, 3]])
```

In [89]:
```
np.transpose(x)
```

Out[89]:
```
array([[0, 2],
       [1, 3]])
```

In [90]:
```
np.random.randint(4,10,size=10)
```

Out[90]:
```
array([8, 5, 7, 7, 9, 5, 4, 7, 4, 9])
```

In [91]:
```
np.zeros((2,3,4))
```

Out[91]:
```
array([[[0., 0., 0., 0.],
        [0., 0., 0., 0.],
        [0., 0., 0., 0.]],

       [[0., 0., 0., 0.],
        [0., 0., 0., 0.],
        [0., 0., 0., 0.]]])
```

An Introduction to Pandas in Python

Pandas is a software library written for the Python programming language. It is used for data manipulation and analysis. It provides special data structures and operations for the manipulation of numerical tables and time series.

Pandas is the name for a Python module, which is rounding up the capabilities of Numpy, Scipy and Matplotlab. The word pandas is an acronym which is derived from "Python and data analysis" and "panel data".

In [92]:
```
#pip install pandas
import pandas as pd
```

Data structures in pandas

Dataframe and series

A DataFrame is a two-dimensional array of values with both a row and a column index.

A Series is a one-dimensional array of values with an index.

	Value
0	NJ
1	CA
2	TX
3	MD
4	OH
5	IL

Column Index

	State	City	Shape
0	NJ	Towaco	Square
1	CA	San Francisco	Oval
2	TX	Austin	Triangle
3	MD	Baltimore	Square
4	OH	Columbus	Hexagon
5	IL	Chicaco	Circle

If it looks like the picture on the left is also present in the picture on the right, you're right! Where a DataFrame is the entire dataset, including all rows and columns — a Series is essentially a single column within that DataFrame.

Series

A Series is a one-dimensional labelled array-like object. It is capable of holding any data type, e.g. integers, floats, strings, Python objects, and so on. It can be seen as a data structure with two arrays: one functioning as the index, i.e. the labels, and the other one contains the actual data

In [93]:
```
import pandas as pd
S = pd.Series([11, 28, 72, 3, 5, 8])
print(S)
```

```
0    11
1    28
2    72
3     3
4     5
5     8
dtype: int64
```

We haven't defined an index in our example, but we see two columns in our output: The right column contains our data, whereas the left column contains the index. Pandas created a default index starting with 0 going to 5, which is the length of the data minus 1.

In [94]:
```
print(S.index)
print(S.values)
```

```
RangeIndex(start=0, stop=6, step=1)
[11 28 72  3  5  8]
```

Difference between Numpy array and Series

There is often some confusion about whether Pandas is an alternative to Numpy, SciPy and Matplotlib. The truth is that it is built on top of Numpy. This means that Numpy is required by pandas. Scipy and Matplotlib on the other hand are not required by pandas but they are extremely useful. That's why the Pandas project lists them as "optional dependency".

In [95]:
```
import numpy as np
X = np.array([11, 28, 72, 3, 5, 8])
print(X)
print(S.values)
# both are the same type:
print(type(S.values), type(X))
```

```
[11 28 72  3  5  8]
[11 28 72  3  5  8]
<class 'numpy.ndarray'> <class 'numpy.ndarray'>
```

In [96]:
```
#What is the actual difference
fruits = ['apples', 'oranges', 'cherries', 'pears'] #We can define Series obj
quantities = [20, 33, 52, 10]
S = pd.Series(quantities, index=fruits)
```

In [97]:
```
print(S)
```

```
apples      20
oranges     33
cherries    52
pears       10
dtype: int64
```

In [98]:
```
#add two series with the same indices, we get a new series with the same inde
fruits = ['apples', 'oranges', 'cherries', 'pears']

S = pd.Series([20, 33, 52, 10], index=fruits)
S2 = pd.Series([17, 13, 31, 32], index=fruits)
print(S + S2)
print("sum of S: ", sum(S))
```

```
apples      37
oranges     46
cherries    83
pears       42
dtype: int64
sum of S:  115
```

In [99]:
```
#The indices do not have to be the same for the Series addition. The index wi
#If an index doesn't occur in both Series, the value for this Series will be
fruits = ['peaches', 'oranges', 'cherries', 'pears']
fruits2 = ['raspberries', 'oranges', 'cherries', 'pears']

S = pd.Series([20, 33, 52, 10], index=fruits)
S2 = pd.Series([17, 13, 31, 32], index=fruits2)
print(S + S2)
```

```
cherries       83.0
oranges        46.0
peaches         NaN
pears          42.0
raspberries     NaN
dtype: float64
```

In [100]:
```
#indices can be completely different, as in the following example.
#We have two indices. One is the Turkish translation of the English fruit nam
fruits = ['apples', 'oranges', 'cherries', 'pears','abc']

fruits_tr = ['elma', 'portakal', 'kiraz', 'armut']

S = pd.Series([20, 33, 52, 10,15], index=fruits)
S2 = pd.Series([17, 13, 31, 32], index=fruits_tr)
print(S + S2)
```

```
abc         NaN
apples      NaN
armut       NaN
cherries    NaN
elma        NaN
kiraz       NaN
oranges     NaN
pears       NaN
portakal    NaN
dtype: float64
```

Series indexing

In [101]:
```
a = [1,2,3,4]
a[0:3]
```

Out[101]:
```
[1, 2, 3]
```

In [102]:
```
S
```

Out[102]:
```
apples      20
oranges     33
cherries    52
pears       10
abc         15
dtype: int64
```

In [103]:
```
print('Single Indexing',S['apples'])
print('@@@@@@@@@@@@@@@@')
print('Multi Indexing ',S[['apples', 'oranges', 'cherries']])
```

```
Single Indexing 20
@@@@@@@@@@@@@@@@
Multi Indexing  apples      20
oranges     33
cherries    52
dtype: int64
```

pandas.Series.apply

The function "func" will be applied to the Series and it returns either a Series or a DataFrame, depending on "func".

Parameter Meaning func a function, which can be a NumPy function that will be applied to the entire Series or a Python function that will be applied to every single value of the series convert_dtype A boolean value. If it is set to True (default), apply will try to find better dtype for elementwise function results. If False, leave as dtype=object args Positional arguments which will be passed to the function "func" additionally to the values from the series. **kwds Additional keyword arguments will be passed as keywords to the function

In [104]:

```
#Ex
print(S)
S.apply(np.log)
```

```
apples      20
oranges     33
cherries    52
pears       10
abc         15
dtype: int64
```

Out[104]:

```
apples      2.995732
oranges     3.496508
cherries    3.951244
pears       2.302585
abc         2.708050
dtype: float64
```

In [105]:

```
S
```

Out[105]:

```
apples      20
oranges     33
cherries    52
pears       10
abc         15
dtype: int64
```

In [106]:

```
def fn:
    if x>50:
        do this
    else:
        do this
```

```
  File "<ipython-input-106-ac87e565d54a>", line 1
    def fn:
          ^
SyntaxError: invalid syntax
```

In [107]:
```
# Let's assume, we have the following task. The test the amount of fruit for
#If there are less than 50 available, we will augment the stock by 10:

S.apply(lambda x: x if x > 50 else x+10 )
```

Out[107]:
```
apples      30
oranges     43
cherries    52
pears       20
abc         25
dtype: int64
```

In [ ]:

In [108]:
```
#Conditioning in a series
S[S>30]
# S>30
```

Out[108]:
```
oranges     33
cherries    52
dtype: int64
```

In [109]:
```
"apples" in S
```

Out[109]:
```
True
```

In [110]:
```
#Creating Series Objects from Dictionaries
cities = {"London":    8615246,
          "Berlin":    3562166,
          "Madrid":    3165235,
          "Rome":      2874038,
          "Paris":     2273305,
          "Vienna":    1805681,
          "Bucharest": 1803425,
          "Hamburg":   1760433,
          "Budapest":  1754000,
          "Warsaw":    1740119,
          "Barcelona": 1602386,
          "Munich":    1493900,
          "Milan":     1350680}

city_series = pd.Series(cities)
print(city_series)
```

```
London       8615246
Berlin       3562166
Madrid       3165235
Rome         2874038
Paris        2273305
Vienna       1805681
Bucharest    1803425
Hamburg      1760433
Budapest     1754000
Warsaw       1740119
Barcelona    1602386
Munich       1493900
Milan        1350680
dtype: int64
```

Handling missing data in pandas

One problem in dealing with data analysis tasks consists in missing data. Pandas makes it as easy as possible to work with missing data.

In [111]:
```
my_cities = ["London", "Paris", "Zurich", "Berlin",
             "Stuttgart", "Hamburg"]

my_city_series = pd.Series(cities,
                           index=my_cities)
my_city_series
```

Out[111]:
```
London       8615246.0
Paris        2273305.0
Zurich             NaN
Berlin       3562166.0
Stuttgart          NaN
Hamburg      1760433.0
dtype: float64
```

Due to the Nan values the population values for the other cities are turned into floats. There is no missing data in the following examples, so the values are int:

In [112]:
```
my_cities = ["London", "Paris", "Berlin", "Hamburg"]

my_city_series = pd.Series(cities,
                           index=my_cities)
my_city_series
```

Out[112]:
```
London     8615246
Paris      2273305
Berlin     3562166
Hamburg    1760433
dtype: int64
```

In [113]:
```
#Finding whether a data is null or not
my_cities = ["London", "Paris", "Zurich", "Berlin",
             "Stuttgart", "Hamburg"]

my_city_series = pd.Series(cities,
                           index=my_cities)
print(my_city_series.isnull())
```

```
London       False
Paris        False
Zurich        True
Berlin       False
Stuttgart     True
Hamburg      False
dtype: bool
```

In [114]:
```
print(my_city_series.notnull())
my_city_series[my_city_series.notnull()]
```

```
London        True
Paris         True
Zurich       False
Berlin        True
Stuttgart    False
Hamburg       True
dtype: bool
```

Out[114]:
```
London     8615246.0
Paris      2273305.0
Berlin     3562166.0
Hamburg    1760433.0
dtype: float64
```

In [115]:
```
#Drop the nulls
print(my_city_series.dropna())
```

```
London     8615246.0
Paris      2273305.0
Berlin     3562166.0
Hamburg    1760433.0
dtype: float64
```

In [116]:
```
#Fill the nulls
print(my_city_series.fillna(10))
```

```
London       8615246.0
Paris        2273305.0
Zurich            10.0
Berlin       3562166.0
Stuttgart         10.0
Hamburg      1760433.0
dtype: float64
```

In [117]:
```
missing_cities = {"Stuttgart":597939, "Zurich":378884}
my_city_series.fillna(missing_cities)
```

Out[117]:
```
London       8615246.0
Paris        2273305.0
Zurich        378884.0
Berlin       3562166.0
Stuttgart     597939.0
Hamburg      1760433.0
dtype: float64
```

In [118]:
```
#Still the values are not integers, we can convert it into int
my_city_series = my_city_series.fillna(0).astype(int)
print(my_city_series)
```

```
London       8615246
Paris        2273305
Zurich             0
Berlin       3562166
Stuttgart          0
Hamburg      1760433
dtype: int32
```

In [119]:
```
ser1 = pd.Series(['a','b','c'])
ser1
```

Out[119]:
```
0    a
1    b
2    c
dtype: object
```

In [120]:
```
ser1.map({'a':'Yes','b':'No','c':'Not sure'})
```

Out[120]:
```
0         Yes
1          No
2    Not sure
dtype: object
```

In [ ]:

In [ ]:

In [ ]: