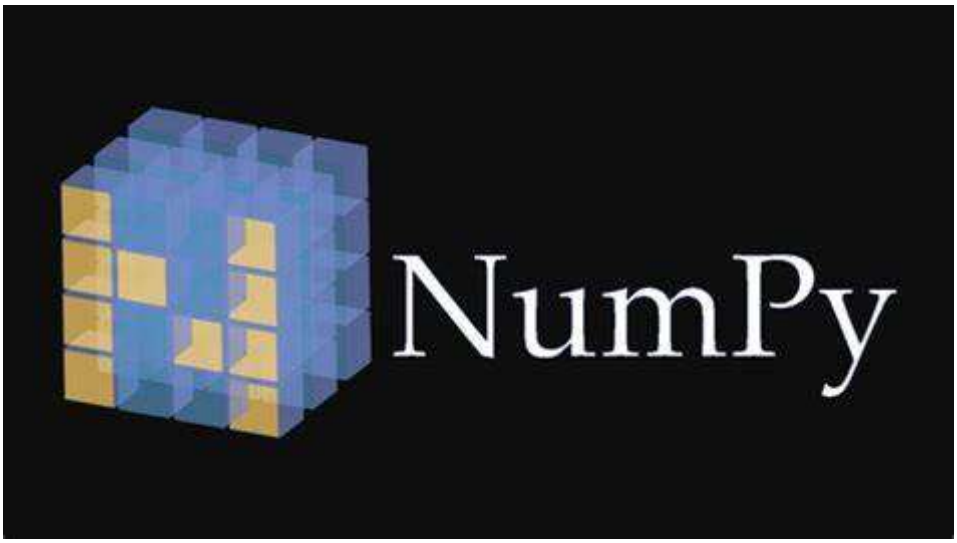


'''NumPy is a Python Library. It stands for 'Numerical Python'. It is a library consisting of multidimensional array objects. '''

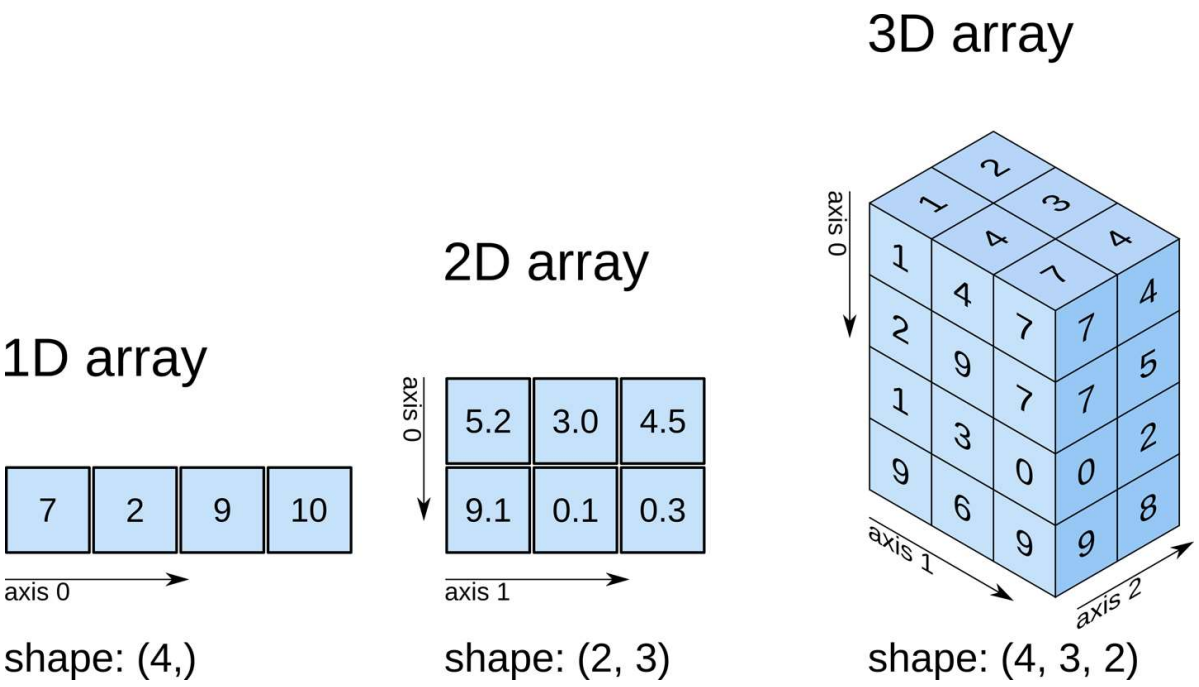
> Numpy array vs List

In [ ]:
```
d='''
Python List:

Lists are the inbuilt data structure of python.
We can store elements of different types in the list.
Items in the list are enclosed between square brackets and separated by commas.
We can even nest lists with other data structures like lists, dictionaries, tuples, et

Python Array:

Arrays are not the in-built data structure readily available in python.
We must import the array from the 'array' or 'numpy' module.
Array stores elements of similar types. If we try to store elements of different types
We can only store elements of the same types in the array.

'''
```

List:

Numpy:

PYTHONHUNT



In [1]:
```
# for installing the numpy
!pip install numpy


#import numpy module with alias np
import numpy as np
```

```
Requirement already satisfied: numpy in c:\users\hi\anaconda3\lib\site-packages (1.2
1.5)
```

In [2]:
```
# Define a numpy array passing a list with  1,2 and 3 as elements in it
a = np.array([1,2,3,4,'a'])
```

In [3]:
```
# print a
a
```

Out[3]:
```
array(['1', '2', '3', '4', 'a'], dtype='<U11')
```

Dimensions in Arrays

Numpy array can be of n dimentions

Lets create arrays of different dimentions.

a=A numpy array with one single integer 10

b=A numpy array passing a list having a list= [1,2,3]

c=A numpy array passing nested list having [[1, 2, 3], [4, 5, 6]] as elements

d=A numpy array passing nested list having [[[1, 2, 3], [4, 5, 6]], [[1, 2, 3], [4, 5, 6]]] as elements

3 points

In [4]:
```
#define a,b,c and d as instructed above

a = np.array(10)
b = np.array([1,2,3])
c = np.array([[1, 2, 3], [4, 5, 6]])
d = np.array([[[1, 2, 3], [4, 5, 6]], [[1, 2, 3], [4, 5, 6]]])
```

Are you ready to check its dimension? Use ndim attribute on each variable to check its dimension

In [5]:
```
#print dimentions of a,b, c and d
print('a dimension:', a.ndim)
print('b dimension:', b.ndim)
print('c dimension:', c.ndim)
print('d dimension:', d.ndim)
```

```
a dimension: 0
b dimension: 1
c dimension: 2
d dimension: 3
```

Lets print there shape as well. You can check shape using shape attribute

In [6]:
```
# print shape of each a,b ,c and d
print(a)
print('shape of a:', a.shape)
print(b)
print('shape of b:', b.shape)
print(c)
print('shape of c:', c.shape)
print(d)
print('shape of d:', d.shape)
```

```
10
shape of a: ()
[1 2 3]
shape of b: (3,)
[[1 2 3]
 [4 5 6]]
shape of c: (2, 3)
[[[1 2 3]
  [4 5 6]]

 [[1 2 3]
  [4 5 6]]]
shape of d: (2, 2, 3)
```

Lets check data type passed in our array. To check data type you can use dtype attribute

In [7]:
```
# print data type of c and d
print(c.dtype)
print(d.dtype)
```

```
int32
int32
```

Lets check the type of our variable. To check type of any numpy variable use type() function

In [8]:
```
#print type of a and b variable
print(type(a))
print(type(b))
```

```
<class 'numpy.ndarray'>
<class 'numpy.ndarray'>
```

In [9]:
```
# Lets check length of array b, using len() function
len(d)
```

Out[9]:
```
2
```

Creating Numpy array

There are multiple ways to create numpy array. Lets walk over them

1. Using arrange() function

8 points

In [10]:
```
#Create a numpy array using arange with 1 and 11 as parameter in it
np.arange(1, 11)
```

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

In [11]:
```
# Create an array using arange passing 1,11 and 2 as parameters
np.arange(1,11,2)
```

Out[11]:
```
array([1, 3, 5, 7, 9])
```

1. Using eye Function

In [12]:
```
# create numpy array using eye function with 3 as passed parameter
np.eye(3, dtype = int)
```

Out[12]:
```
array([[1, 0, 0],
       [0, 1, 0],
       [0, 0, 1]])
```

Wohoo! eye return a 2-D array with ones on the diagonal and zeros elsewhere.

1. Using zero function

In [13]:
```
#create a numpy array using zero function with (3,2) as passed parameter takes default
np.zeros((3, 5),dtype='int') # default float
```

Out[13]:
```
array([[0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0]])
```

Zero function returns a new array of given shape and type, filled with zeros.

1. Using ones Function

In [14]:
```
#create a numpy array using ones function with (3,2) as passed parameter
np.ones((6, 3),dtype='int')
```

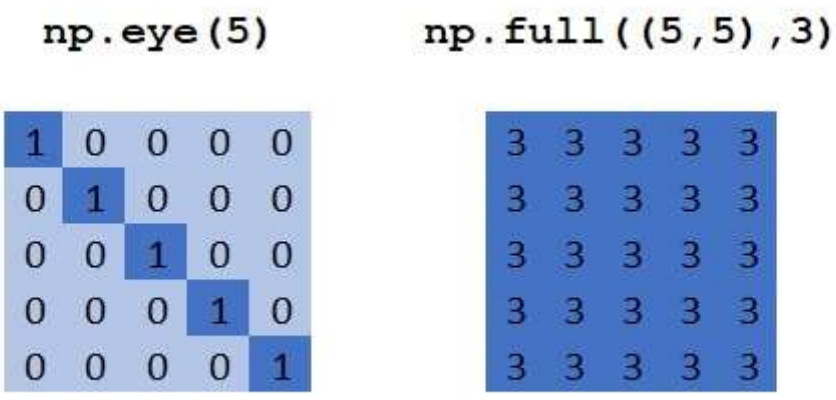
Out[14]:
```
array([[1, 1, 1],
       [1, 1, 1],
       [1, 1, 1],
       [1, 1, 1],
       [1, 1, 1],
       [1, 1, 1]])
```

You noticed! ones function returns a new array of given shape and type, filled with ones.

1. Using full Function

In [15]:
```
#create a numpy array using full function with (3,2) and 2 as passed parameter
np.full((2, 3), 10,dtype=int)
```

Out[15]:
```
array([[10, 10, 10],
       [10, 10, 10]])
```



1. Using diag function

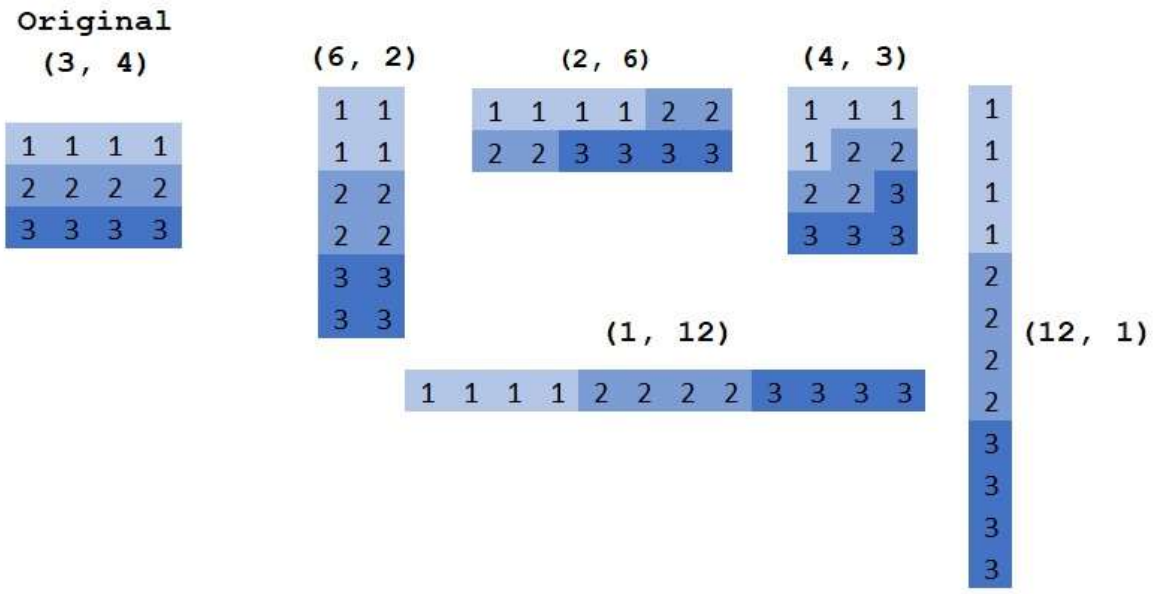
In [16]:
```
#create a numpy array using diag function passing a list [1,2,3,4,5]
x = [1,2,3,4,5,6,7]
np.diag(x)
```

Out[16]:
```
array([[1, 0, 0, 0, 0, 0, 0],
       [0, 2, 0, 0, 0, 0, 0],
       [0, 0, 3, 0, 0, 0, 0],
       [0, 0, 0, 4, 0, 0, 0],
       [0, 0, 0, 0, 5, 0, 0],
       [0, 0, 0, 0, 0, 6, 0],
       [0, 0, 0, 0, 0, 0, 7]])
```

Numpy Reshape

Reshaping means changing the shape of an array. The shape of an array is the number of elements in each dimension. By reshaping we can add or remove dimensions or change number of elements in each dimension.

6 points



In [17]:
```
# Using arange() to generate numpy array x with numbers between 1 to 16
x = np.arange(1, 17)
print(x)
print(x.shape)
print(x.ndim)
```

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

In [18]:
```
# Reshape x with 2 rows and 8 columns
n=x.reshape((8,2)) # 1d
print(n)
```

```
print(n.shape)
print(n.ndim)
```

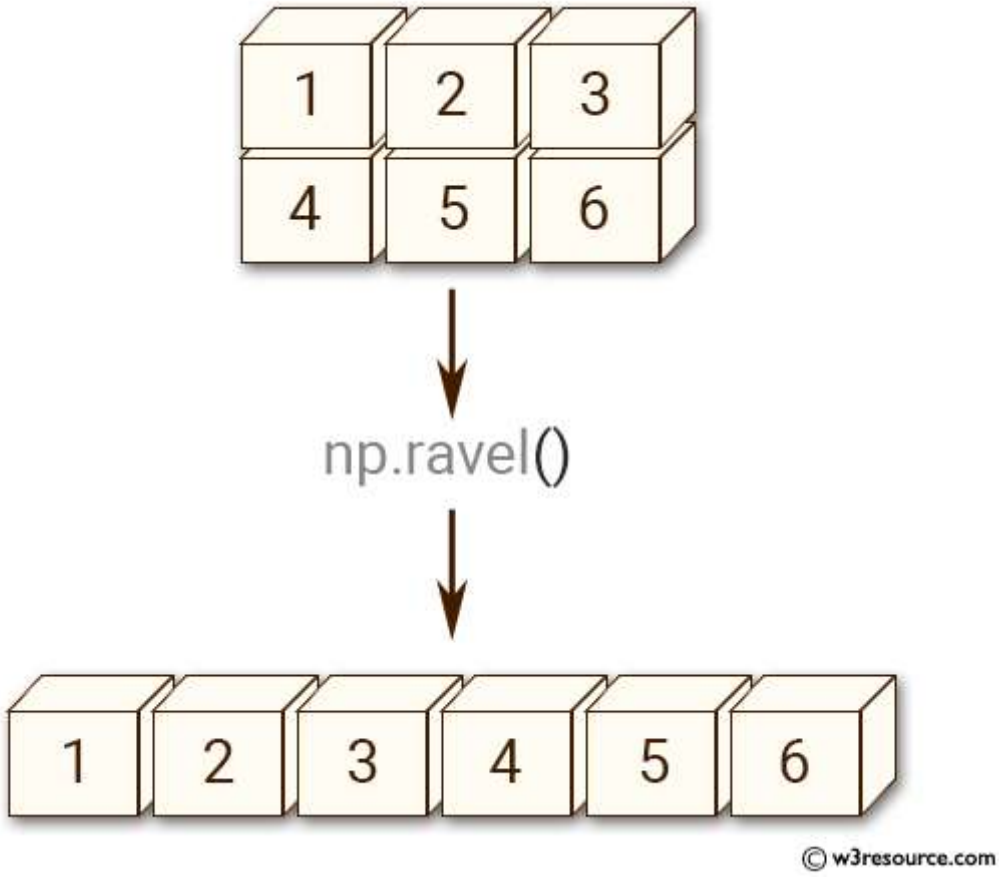
```
[[ 1  2]
 [ 3  4]
 [ 5  6]
 [ 7  8]
 [ 9 10]
 [11 12]
 [13 14]
 [15 16]]
(8, 2)
2
```

In [19]:
```
n1=n.reshape((4,2,2))
print(n1)
print(n1.shape)
print(n1.ndim)
```

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

 [[ 5  6]
  [ 7  8]]

 [[ 9 10]
  [11 12]]

 [[13 14]
  [15 16]]]
(4, 2, 2)
3
```

In [20]:
```
# convert to old dimension
v=n.ravel()
print(v.shape)
print(v)
print(v.ndim)
```

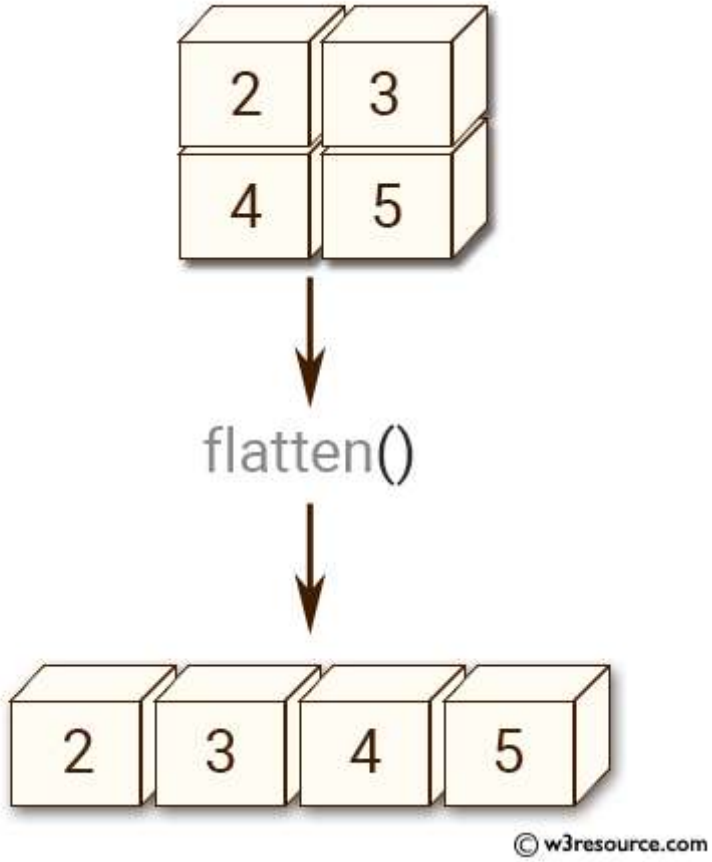
```
(16,)
[ 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16]
1
```



In [21]:
```
# reshape x with dimension that will have 2 arrays that contains 4 arrays, each with 2
print(x)
x.reshape((2,8))
```

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

Out[21]:
```
array([[ 1,  2,  3,  4,  5,  6,  7,  8],
       [ 9, 10, 11, 12, 13, 14, 15, 16]])
```



In [22]:
```
# Flattening x convert array into one dimensional
x.flatten()
```

Out[22]:
```
array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16])
```

NumPy Array Indexing

Array indexing is the same as accessing an array element. You can access an array element by referring to its index number.

The indexes in NumPy arrays start with 0, meaning that the first element has index 0, and the second has index 1 etc.

4 points

In [23]:
```
# Create an array a with all even numbers between 1 to 17
a = np.arange(2,17,2)

# print a
a
```

Out[23]:
```
array([ 2,  4,  6,  8, 10, 12, 14, 16])
```

In [24]:
```
# Get third element in array a
a[2]
```

Out[24]:
```
6
```

In [25]:
```
#Print 3rd, 5th, and 7th element in array a
a[0:7:2]
```

Out[25]:
```
array([ 2,  6, 10, 14])
```

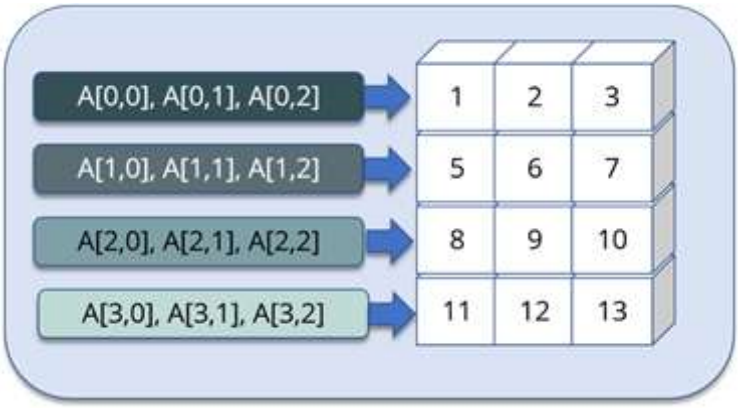
Lets check the same for 2 D array

In [26]:
```
# Define an array 2-D a with [[1,2,3],[4,5,6],[7,8,9]] as its elements.
a = np.array([[1,2,3],[4,5,6],[7,8,9]])
a
```

Out[26]:
```
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

In [27]:
```
# print the 3rd element from the 3rd row of a
a[2,2]
```

Out[27]:
```
9
```



In [28]:
```
# Define an array b again with [[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]] as
b = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])
b.shape
b
```

Out[28]:
```
array([[[ 1,  2,  3],
        [ 4,  5,  6]],

       [[ 7,  8,  9],
        [10, 11, 12]]])
```

In [29]:
```
# Print 3rd element from 2nd list which is 1st list in nested list passed. Confusing r
b[1][1][2]
```

Out[29]:
```
12
```

Great job! So now you have learned how to to indexing on various dimentions of numpy array.

NumPy Array Slicing

Slicing in python means taking elements from one given index to another given index.

1. We pass slice instead of index like this: [start:end].
2. We can also define the step, like this: [start:end:step].
3. If we don't pass start its considered 0
4. If we don't pass end its considered length of array in that dimension
5. If we don't pass step its considered 1

5 points

1. **Array slicing in 1-D array.**

In [30]:
```
arr=np.arange(1,11)
arr
```

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

In [31]:
```
# Slice elements from 1st to 5th element from array arr:
arr[0:5]
```

Out[31]:
```
array([1, 2, 3, 4, 5])
```

In [32]:
```
# Slice elements from index 5 to the end of the array arr:
arr[5:]
```

Out[32]:
```
array([ 6,  7,  8,  9, 10])
```

In [33]:
```
# Slice elements from the beginning to index 5 (not included) in array arr:
arr[0:5]
```

Out[33]:
```
array([1, 2, 3, 4, 5])
```

In [34]:
```
# Print every other element from index 1 to index 7:
arr[1:7:2]
```

Out[34]:
```
array([2, 4, 6])
```

In [35]:
```
# Return every other element from the entire array arr:
arr[0:10:2]
```

Out[35]:
```
array([1, 3, 5, 7, 9])
```

In [36]:
```
# Print array a
a
```

Out[36]:
```
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

In [37]:
```
# In array 'a' print index 2 from all the rows :
arr[2]
```

Out[37]:
```
3
```

In [38]:
```
# From all the elements in 'a', slice index 1 till end, this will return a 2-D array:
a[0:,1:]
```

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

Hurray! You have learned Slicing in Numpy array. Now you know to access any numpy array.

Numpy copy vs view

7 points

In [39]:
```
x1= np.arange(10)
x1
```

Out[39]:
```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

In [40]:
```
# assign x2 = x1
x2 = x1 # view
```

In [41]:
```
#print x1 and x2
print(x1)
print(x2)
```

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

Ok now you have seen that both of them are same

In [42]:
```
# change 1st element of x2 as 10
x2[0] = 10
```

In [43]:
```
#Again print x1 and x2
print(x1)
print(x2)
```

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

In [44]:
```
print(id(x1))
print(id(x2))
```

```
2143762988176
2143762988176
```

Wait a minute. Just check your above result on change of x2, x1 also got changed. Why?

Lets check if both the variables shares memory. Use numpy shares_memory() function to check if both x1 and x2 shares a memory.

In [45]:
```
# Check memory share between x1 and x2
print(np.shares_memory(x1, x2))
```

```
print(id(x1))
print(id(x2))
```

```
True
2143762988176
2143762988176
```

Hey It's True they both share memory

Shall we try **view()** function also likwise.

In [46]:
```
# Create a view of x1 and store it in x3.
x3 = x1.view()
```

In [47]:
```
# Again check memory share between x1 and x3
np.shares_memory(x1, x3)
```

Out[47]:
```
True
```

In [48]:
```
#Change 1st element of x3=100
x3[0] = 100
```

In [49]:
```
#print x1 and x3 to check if changes reflected in both
print(x1)
print(x3)
```

```
[100   1   2   3   4   5   6   7   8   9]
[100   1   2   3   4   5   6   7   8   9]
```

Now its proved.

Lets see how **Copy()** function works

In [50]:
```
# Now create an array x4 which is copy of x1

x4 = np.copy(x1)
```

In [51]:
```
# Change the last element of x4 as 900
x4[-1] = 900
```

In [52]:
```
# print both x1 and x4 to check if changes reflected in both
print(x1)
print(x4)
```

```
[100   1   2   3   4   5   6   7   8   9]
[100   1   2   3   4   5   6   7   8 900]
```

Hey! such an intresting output. You noticed buddy! your original array didn't get changed on change of its copy ie. x4.

In [53]:
```
#Check memory share between x1 and x4
print(id(x1))
print(id(x4))
np.shares_memory(x1, x4)
```

```
2143762988176
2143763076944
```

Out[53]:
```
False
```

You see! x1 and x4 don't share its memory.

So with all our outputs we can takeaway few points:

1. The main difference between a copy and a view of an array is that the copy is a new array, and the view is just a view of the original array.

2. The copy owns the data and any changes made to the copy will not affect original array, and any changes made to the original array will not affect the copy.

3. The view does not own the data and any changes made to the view will affect the original array, and any changes made to the original array will affect the view.

More operations on Numpy

1. Applying conditions

5 points

In [54]:
```
#print a
a
```

Out[54]:
```
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

We are going to use 'a' array for all our array condition operations.

In [55]:
```
# Check if every element in array a greater than 3 or not Using '>' notation
a > 3
```

Out[55]:
```
array([[False, False, False],
       [ True,  True,  True],
       [ True,  True,  True]])
```

In [56]:
```
# Get a list with all elements of array 'a' grater than 3
a[a > 3]
```

Out[56]:
```
array([4, 5, 6, 7, 8, 9])
```

In [57]:
```
# Get a list with all elements of array 'a' greater than 3 but less than 6
a[(a > 3) & (a < 6)]
```

Out[57]:
```
array([4, 5])
```

In [58]:
```
# check if each elements in array 'x1' equals array 'x4' using '==' notation
x1 == x4
# print(x1)
# print(x4)
```

Out[58]:
```
array([ True,  True,  True,  True,  True,  True,  True,  True,  True,
       False])
```

You can see in above output that the last element is not same in both x1 and x4

Well done so far.

Lets check how to transpose an array

2. Transposing array

In [59]:
```
# Print Transpose of array 'a'
print(a.transpose())
#print array 'a'
print("-----------------")
print(a)
```

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

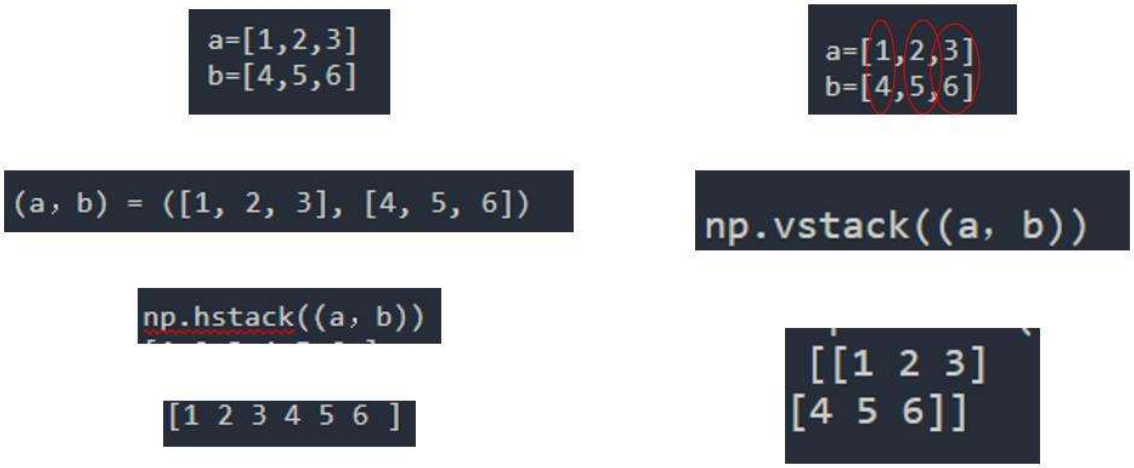
In above output all the rows became columns by transposing

3. hstack vs vstack function

Stacking is same as concatenation, the only difference is that stacking is done along a new axis.

NumPy provides a helper function:

1. hstack() to stack along rows.
2. vstack() to stack along columns



In [60]:
```
# stack x1 and x4 along columns.
print(x1)
print(x4)
np.vstack((x1, x4))
```

Out[60]:
```
[100   1   2   3   4   5   6   7   8   9]
[100   1   2   3   4   5   6   7   8 900]
array([[100,   1,   2,   3,   4,   5,   6,   7,   8,   9],
       [100,   1,   2,   3,   4,   5,   6,   7,   8, 900]])
```

In [61]:
```
#stack x1 and x4 along rows
np.hstack((x1, x4))
```

Out[61]:
```
array([100,   1,   2,   3,   4,   5,   6,   7,   8,   9, 100,   1,   2,
         3,   4,   5,   6,   7,   8, 900])
```

Adding, Insert and delete Numpy array

3 points

Lets use insert() function which Inserts values into array before specified index value

In [62]:
```
# Inserts values into array x1 before index 4 with elements of x4
print(x1)
print(x4)
np.insert(x1, 4, x4)
```

Out[62]:
```
[100   1   2   3   4   5   6   7   8   9]
[100   1   2   3   4   5   6   7   8 900]
array([100,   1,   2,   3, 100,   1,   2,   3,   4,   5,   6,   7,   8,
       900,   4,   5,   6,   7,   8,   9])
```

You can see in above output we have inserted all the elements of x4 before index 4 in array x1.

In [63]:
```
# delete 2nd element from array x2
print(x2)
np.delete(x2, 0)
```

Out[63]:
```
[100   1   2   3   4   5   6   7   8   9]
array([1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Did you see? 2 value is deleted from x2 which was at index position 2

Mathmatical operations on Numpy array

8 points

In [64]:
```
#defining a
import numpy as np
a= np.array([[1,2,3],[4,5,6],[7,8,9]])
```

In [65]:
```
# print trigonometric sin value of each element of a
np.sin(a)
```

Out[65]:
```
array([[ 0.84147098,  0.90929743,  0.14112001],
       [-0.7568025 , -0.95892427, -0.2794155 ],
       [ 0.6569866 ,  0.98935825,  0.41211849]])
```

In [66]:
```
np.sin(0)
```

Out[66]:
```
0.0
```

In [67]:
```
np.tan(0)
```

Out[67]:
```
0.0
```

In [68]:
```
(np.sin(a))/(np.cos(a))
```

Out[68]:
```
array([[ 1.55740772, -2.18503986, -0.14254654],
       [ 1.15782128, -3.38051501, -0.29100619],
       [ 0.87144798, -6.79971146, -0.45231566]])
```

In [69]:
```
# print trigonometric cos value of each element of a
print(np.cos(a))
print(np.tan(a))
```

```
[[ 0.54030231 -0.41614684 -0.9899925 ]
 [-0.65364362  0.28366219  0.96017029]
 [ 0.75390225 -0.14550003 -0.91113026]]
[[ 1.55740772 -2.18503986 -0.14254654]
 [ 1.15782128 -3.38051501 -0.29100619]
 [ 0.87144798 -6.79971146 -0.45231566]]
```

In [70]:
```
# Print exponential value of each elements of a
np.exp(a) # **
```

Out[70]:
```
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 [71]:
```
# print total sum of elements of a
np.sum(a)
```

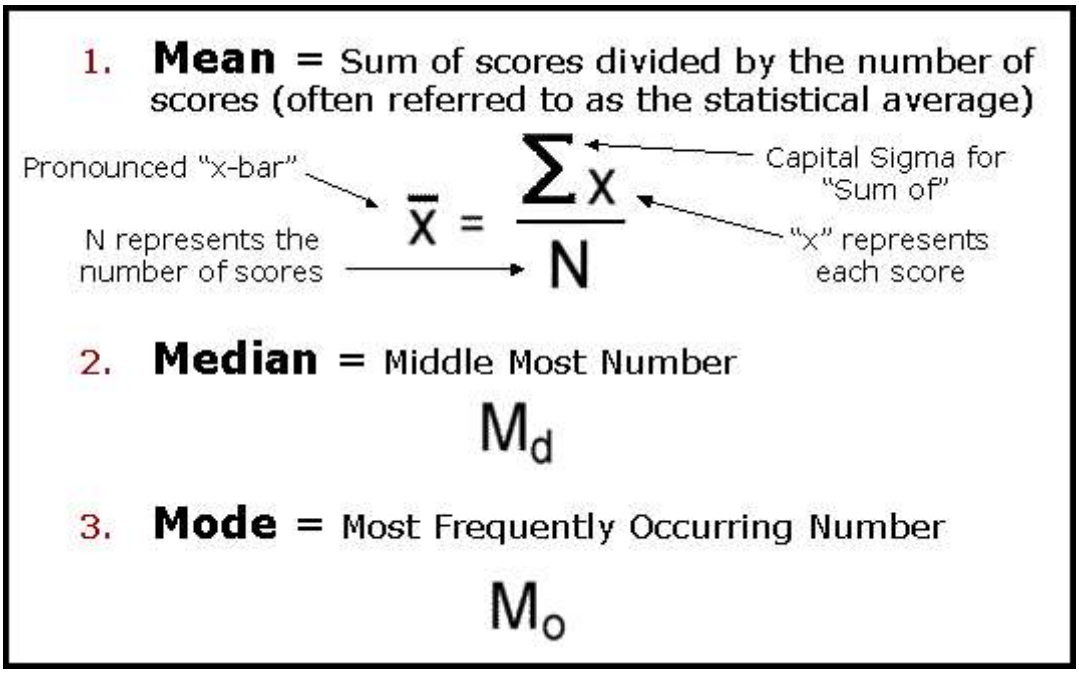
Out[71]:
```
45
```

In [72]:
```
# Print sum in array a column wise
np.sum(a, axis=1)
```

Out[72]:
```
array([ 6, 15, 24])
```

In [73]:
```
# Print sum in array a row wise
np.sum(a, axis=0)
```

Out[73]:
```
array([12, 15, 18])
```

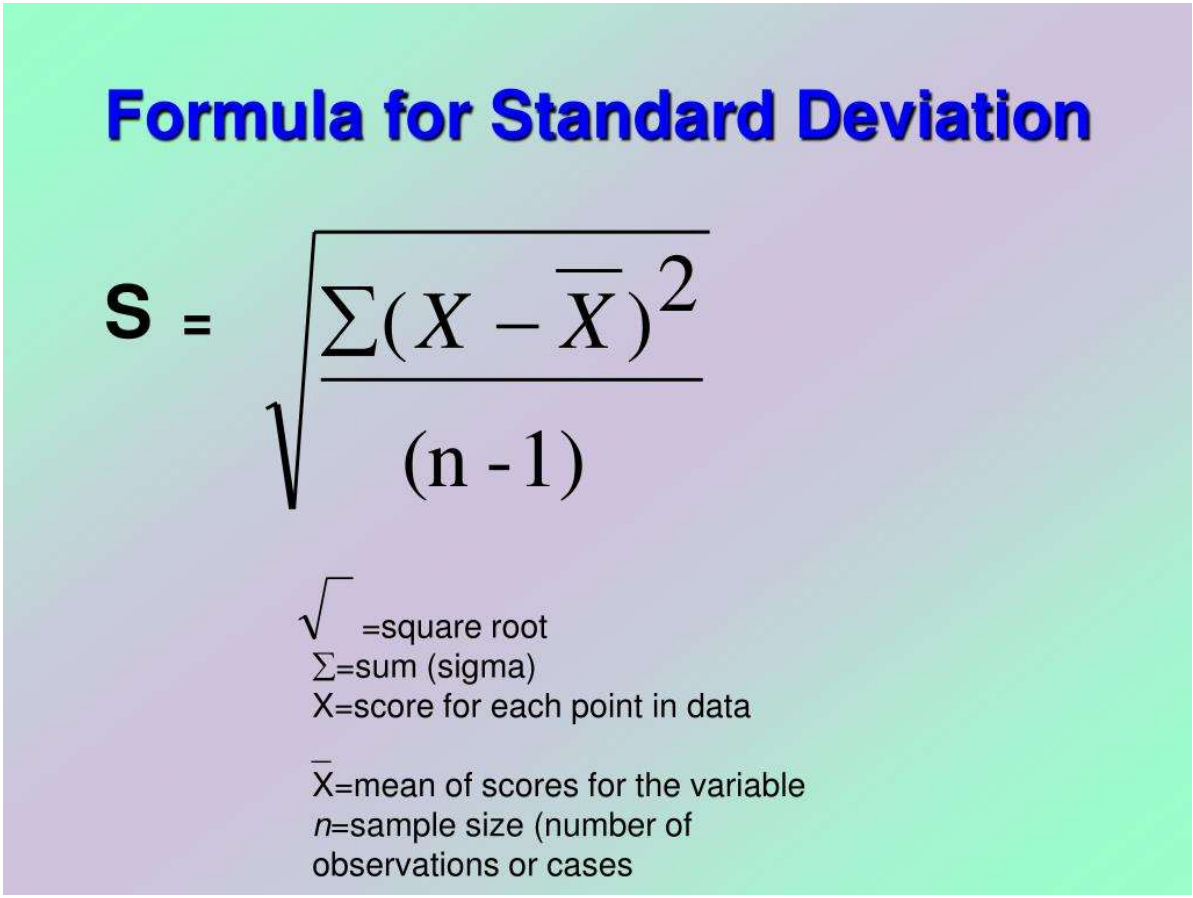


In [74]:
```
# print median of array a
np.median(a)
```

Out[74]:
```
5.0
```

In [75]:
```
# print mean of array a
np.mean(a)
```

Out[75]:
```
5.0
```



In [76]:
```
# print standard deviation of array a
np.std(a)
```

Out[76]:
```
2.581988897471611
```

In [77]:
```
#print largest element present in array a
np.max(a)
```

Out[77]:
```
9
```

In [78]:
```
# print sorted x4 array
np.sort(x4)
```

Out[78]:
```
array([  1,   2,   3,   4,   5,   6,   7,   8, 100, 900])
```

Searching Arrays

You can search an array for a certain value, and return the indexes that get a match.

To search an array, use the where() method.

4 points

In [79]:
```
# Create a numpy array
arr = np.array([[12,14,17,19,24,27,35,38]])
```

```
# Use numpy.where() with 1 Dimensional Arrays
arr1 = np.where(arr > 17)
print(arr1)
```

```
(array([0, 0, 0, 0, 0], dtype=int64), array([3, 4, 5, 6, 7], dtype=int64))
```

In [ ]:

In [80]:
```
print(dir(list))
```

```
['__add__', '__class__', '__class_getitem__', '__contains__', '__delattr__', '__delit
em__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '_
_getitem__', '__gt__', '__hash__', '__iadd__', '__imul__', '__init__', '__init_subcla
ss__', '__iter__', '__le__', '__len__', '__lt__', '__mul__', '__ne__', '__new__', '__
reduce__', '__reduce_ex__', '__repr__', '__reversed__', '__rmul__', '__setattr__', '_
_setitem__', '__sizeof__', '__str__', '__subclasshook__', 'append', 'clear', 'copy',
'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort']
```

In [81]:
```
print(dir(set))
```

```
['__and__', '__class__', '__class_getitem__', '__contains__', '__delattr__', '__dir_
_', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__gt__', '__has
h__', '__iand__', '__init__', '__init_subclass__', '__ior__', '__isub__', '__iter__',
'__ixor__', '__le__', '__len__', '__lt__', '__ne__', '__new__', '__or__', '__rand__',
'__reduce__', '__reduce_ex__', '__repr__', '__ror__', '__rsub__', '__rxor__', '__seta
ttr__', '__sizeof__', '__str__', '__sub__', '__subclasshook__', '__xor__', 'add', 'cl
ear', 'copy', 'difference', 'difference_update', 'discard', 'intersection', 'intersec
tion_update', 'isdisjoint', 'issubset', 'issuperset', 'pop', 'remove', 'symmetric_dif
ference', 'symmetric_difference_update', 'union', 'update']
```

In [82]:
```
import keyword
print(keyword.kwlist)
print(len(keyword.kwlist))
```

```
['False', 'None', 'True', '__peg_parser__', 'and', 'as', 'assert', 'async', 'await',
'break', 'class', 'continue', 'def', 'del', 'elif', 'else', 'except', 'finally', 'fo
r', 'from', 'global', 'if', 'import', 'in', 'is', 'lambda', 'nonlocal', 'not', 'or',
'pass', 'raise', 'return', 'try', 'while', 'with', 'yield']
36
```

In [ ]:

In [ ]:

In [ ]: