Numpy Basics

Welcome to section of Numpy. This is one of the the most used Python libraries for data science. NumPy consists of a powerful data structure called multidimensional arrays. Pandas is another powerful Python library that provides fast and easy data analysis platform.

NumPy is a library written for scientific computing and data analysis. It stands for numerical python and also known as array oriented computing.

The most basic object in NumPy is the ndarray, or simply an array which is an n-dimensional, homogeneous array. By homogenous, we mean that all the elements in a NumPy array have to be of the same data type, which is commonly numeric (float or integer).

Why Numpy?

convenience & speed

Numpy is much faster than the standard python ways to do computations.

Vectorised code typically does not contain explicit looping and indexing etc. (all of this happens behind the scenes, in precompiled C-code), and thus it is much more concise.

Also, many Numpy operations are implemented in C which is basically being executed behind the scenes, avoiding the general cost of loops in Python, pointer indirection and per-element dynamic type checking. The speed boost depends on which operations you're performing.

NumPy arrays are more compact than lists, i.e. they take much lesser storage space than lists

***Let's get started with our Numpy Assigment**

2 points

You can check this numpy video too! : https://www.youtube.com/watch?v=QUT1VHiLmmI (https://www.youtube.com/watch?v=QUT1VHiLmmI)

In [1]:

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

We can create a NumPy ndarray object by using the array() function. To create an ndarray, we can pass a list, tuple or any array-like object into the array() method, and it will be converted into an ndarray:

In [2]:

```
# Define a numpy array passing a list with  1,2 and 3 as elements in it
a =
```

In [3]:

```
# print a
a
```

Out[3]:

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

Dimensions in Arrays

Reference: https://www.youtube.com/watch?v=BNAfVruKKkU (https://www.youtube.com/watch?v=BNAfVruKKkU)

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 =
b =
c =
d =
```

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

In [5]:

```
#print dimentions of a,b, c and d
```

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

Hey hey. Did you see! you have created 0-D,1-DeprecationWarning, 2-D and 3-D arrays.

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

In [6]:

```
# print shape of each a,b ,c and d
```

```
shape of a: ()
shape of b: (3,)
shape of c: (2, 3)
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
```

```
int32
int32
```

Above output mean our array is having int type elements in it.

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
```

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

In [9]:

```
# Lets check length of array b, using len() function
```

Out[9]:

```
3
```

Bravo!You have Defined ndarray i.e numpy array in variable a nd b. Also you have successfully learned how to create numpy.

Performance measurement

I mentioned that the key advantages of numpy are convenience and speed of computation.

You'll often work with extremely large datasets, and thus it is important point for you to understand how much computation time (and memory) you can save using numpy, compared to standard python lists.

2 points

Create two list l1 and l2 where, l1=[10,20,30] and l2=[40,50,60] Also define two numpy arrays l3,l4 where l3 has l1 as element and l4 has l2 as element

In [10]:

```
# Define l1,l2,l3 and l4 as stated above.
l1 =
l2 =
l3 =
l4 =
```

Lets multiply each elements of l1 with corresponding elements of l2

Here use list comprehention to do so. Lets see how much you remember your work in other assignments.

Note: use %timeit as prefix before your line of code inorder to calculate total time taken to run that line
eg. %timeit my_code

In [11]:

```
#code here as instructed above
%timeit #code here
```

```
1.6 µs ± 248 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)
```

Lets mulptiply l3 and l4

Note: use %timeit as prefix before your line of code inorder to calculate total time taken to run that line

In [12]:

```
%timeit #code here
```

```
1.31 µs ± 117 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)
```

Don't worry if still your one line of code is running. Its because your system is calculating total time taken to run your code.

Did you notice buddy! time taken to multiply two lists takes more time than multiplyimg two numpy array. Hence proved that numpy arrays are faster than lists.

Fun Fact time!:

You know in many data science interviews it is asked that what is the difference between list and array.
The answer is: https://www.youtube.com/watch?v=XI6PHo_gP4E (https://www.youtube.com/watch?v=XI6PHo_gP4E)

so in numpy arrays I can do everything without even writing a loop? yes... ohh wao

Creating Numpy array

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

1. Using arrange() function
   Refer: https://numpy.org/doc/stable/reference/generated/numpy.arange.html (https://numpy.org/doc/stable/reference/generated/numpy.arange.html)

8 points

In [13]:

```
#Create a numpy array using arange with 1 and 11 as parameter in it
```

Out[13]:

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

This means using arrange we get evenly spaced values within a given interval. Interval? Yes you can mention interval as well as third parameter in it.

In [14]:

```
# Create an array using arange passing 1,11 and 2 as parameters
```

Out[14]:

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

Did you see? you got all odd numbers as you had mentioned interval between 1 and 10. Also note that 11 is excluded and hence arrange function counted till 10

2. Using eye Function
   Refer: https://numpy.org/devdocs/reference/generated/numpy.eye.html (https://numpy.org/devdocs/reference/generated/numpy.eye.html)

In [15]:

```
# create numpy array using eye function with 3 as passed parameter
```

Out[15]:

```
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.

3. Using zero function
   Refer: https://numpy.org/doc/stable/reference/generated/numpy.zeros.html (https://numpy.org/doc/stable/reference/generated/numpy.zeros.html)

In [16]:

```
#create a numpy array using zero function with (3,2) as passed parameter
```

Out[16]:

```
array([[0., 0.],
       [0., 0.],
       [0., 0.]])
```

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

4. Using ones Function
   Refer: https://numpy.org/doc/stable/reference/generated/numpy.ones.html (https://numpy.org/doc/stable/reference/generated/numpy.ones.html)

In [17]:

```
#create a numpy array using ones function with (3,2) as passed parameter
```

Out[17]:

```
array([[1., 1.],
       [1., 1.],
       [1., 1.]])
```

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

5. Using full Function
   Refer: https://numpy.org/doc/stable/reference/generated/numpy.full.html (https://numpy.org/doc/stable/reference/generated/numpy.full.html)

In [18]:

```
#create a numpy array using full function with (3,2) and 2 as passed parameter
```

Out[18]:

```
array([[2, 2, 2],
       [2, 2, 2],
       [2, 2, 2]])
```

Yeah! full function return a new array of given shape and type, filled with fill_value, here it is 2

6. Using diag function
   Refer: https://numpy.org/doc/stable/reference/generated/numpy.diag.html (https://numpy.org/doc/stable/reference/generated/numpy.diag.html)

In [19]:

```
#create a numpy array using diag function passing a list [1,2,3,4,5]
```

Out[19]:

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

Oh yeah! diag function extract a diagonal or construct a diagonal array.

7. Using tile function
   Refer: https://numpy.org/doc/stable/reference/generated/numpy.tile.html (https://numpy.org/doc/stable/reference/generated/numpy.tile.html)

In [20]:

```
# Create a numpy array v with [1,2,3] as its elements
v =

#Use tile function of numpy and pass v and (3,1) as its parametrs
```

Out[20]:

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

Returns an array by repeating an input array the number of times given by mentioned shape. Here you can see that you stacked 3 copies of v on top of each other

8. Using linspace Function
   Refer: https://numpy.org/doc/stable/reference/generated/numpy.linspace.html (https://numpy.org/doc/stable/reference/generated/numpy.linspace.html)

In [21]:

```
# Create an array with 100 values between 1 and 50 using linspace
```

Out[21]:

```
array([ 1.        ,  1.49494949,  1.98989899,  2.48484848,  2.97979798,
        3.47474747,  3.96969697,  4.46464646,  4.95959596,  5.45454545,
        5.94949495,  6.44444444,  6.93939394,  7.43434343,  7.92929293,
        8.42424242,  8.91919192,  9.41414141,  9.90909091, 10.4040404 ,
       10.8989899 , 11.39393939, 11.88888889, 12.38383838, 12.87878788,
       13.37373737, 13.86868687, 14.36363636, 14.85858586, 15.35353535,
       15.84848485, 16.34343434, 16.83838384, 17.33333333, 17.82828283,
       18.32323232, 18.81818182, 19.31313131, 19.80808081, 20.3030303 ,
       20.7979798 , 21.29292929, 21.78787879, 22.28282828, 22.77777778,
       23.27272727, 23.76767677, 24.26262626, 24.75757576, 25.25252525,
       25.74747475, 26.24242424, 26.73737374, 27.23232323, 27.72727273,
       28.22222222, 28.71717172, 29.21212121, 29.70707071, 30.2020202 ,
       30.6969697 , 31.19191919, 31.68686869, 32.18181818, 32.67676768,
       33.17171717, 33.66666667, 34.16161616, 34.65656566, 35.15151515,
       35.64646465, 36.14141414, 36.63636364, 37.13131313, 37.62626263,
       38.12121212, 38.61616162, 39.11111111, 39.60606061, 40.1010101 ,
       40.5959596 , 41.09090909, 41.58585859, 42.08080808, 42.57575758,
       43.07070707, 43.56565657, 44.06060606, 44.55555556, 45.05050505,
       45.54545455, 46.04040404, 46.53535354, 47.03030303, 47.52525253,
       48.02020202, 48.51515152, 49.01010101, 49.50505051, 50.        ])
```

Wao! linspace returns evenly spaced numbers over a specified interval.

Hey but you saw some similar defination for arrange function

The main difference both of them is that arange return values with in a range which has a space between values (in other words the step) and linspace returns set of samples with in a given interval.

Numpy Random numbers

Fun Fact:

You can create a numpy array with random numbers also. How? Uisng random function
Lets see how Refer: https://numpy.org/doc/stable/reference/random/generated/numpy.random.rand.html (https://numpy.org/doc/stable/reference/random/generated/numpy.random.rand.html)

3 points

In [22]:

```
# Generate one random number between 0 and 1 using numpy's random.rand() function.
```

Out[22]:

```
0.9629715603545379
```

Run the above cell again and check if number changes.

Yeah it changes. That's so random :)

In [23]:

```
# so let say I want a random value between 2 and 50
```

Out[23]:

```
22.429818654925413
```

Run the above cell again and check if number changes and its between 2 to 50.

Now lets create an array with random numbers 0 to 1 of shape 3X3

In [24]:

```
#get an array as stated above
```

Out[24]:

```
array([[0.47070116, 0.88920761, 0.19064915],
       [0.21150292, 0.52151531, 0.42883075],
       [0.27538653, 0.7992213 , 0.49010051]])
```

Smile! you got it how to create a numpy array with random numbers.

Numpy Reshape

Reference: https://www.youtube.com/watch?v=sGCuryS8zjc (https://www.youtube.com/watch?v=sGCuryS8zjc)

reference doc: https://numpy.org/doc/stable/reference/generated/numpy.reshape.html (https://numpy.org/doc/stable/reference/generated/numpy.reshape.html)

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 [25]:

```
# Using arange() to generate numpy array x with numbers between 1 to 16
x=
```

So here x is our 1-D array along with being sweet sixteen array ;). Lets reshape our x into 2-D and 3-D array using Reshape

1. Reshaping 1-D to 2-D

In [26]:

```
# Reshape x with 2 rows and 8 columns
```

Out[26]:

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

As you can see above that our x changed into 2D matrix

2. Reshaping 1-D to 3-D array

In [27]:

```
# reshape x with dimension that will have 2 arrays that contains 4 arrays, each with 2 elem
```

Out[27]:

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

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

Fun Fact:

<u>Unknown Dimension</u>

You are allowed to have one "unknown" dimension.Meaning that you do not have to specify an exact number for one of the dimensions in the reshape method. Pass -1 as the value, and NumPy will calculate this number for you. Awesome right?

In [28]:

```
# Use unknown dimention to reshape x into 2-D numpy array with shape 4*4
```

Out[28]:

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

In [29]:

```
# Use unknown dimention to  reshape x into 3-D numpy array with 2 arrays that contains 4 ar
y=

# print y
print(y)
```

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

 [[ 9 10]
  [11 12]
  [13 14]
  [15 16]]]
```

Note: We can not pass -1 to more than one dimension.

Another cool Fact: -1 can be used to flatten an array which means converting a multidimensional array into a 1D array.

```
Lets apply this technique on y which is 3-D array
```

In [30]:

```
# Flattening y
```

Out[30]:

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

Awesome work!

NumPy Array Indexing

Reference: https://www.youtube.com/watch?v=bFv66_RXLb4 (https://www.youtube.com/watch?v=bFv66_RXLb4)

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 [31]:

```
# Create an array a with all even numbers between 1 to 17
a =

# print a
a
```

Out[31]:

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

In [32]:

```
# Get third element in array a
```

Out[32]:

```
6
```

In [33]:

```
#Print 3rd, 5th, and 7th element in array a
```

Out[33]:

```
array([ 6, 10, 14])
```

Lets check the same for 2 D array

In [34]:

```
# Define an array 2-D a with [[1,2,3],[4,5,6],[7,8,9]] as its elements.
```

In [35]:

```
# print the 3rd element from the 3rd row of a
```

Out[35]:

```
9
```

Well done!

Now lets check indexing for 3 D array

In [36]:

```
# Define an array b again with [[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]] as its e
b =
```

In [37]:

```
# Print 3rd element from 2nd list which is 1st list in nested list passed. Confusing right?
```

Out[37]:

```
6
```

Well done.!

Have you heared about **negative indexing?**

We can use negative indexing to access an array from the end.

In [38]:

```
# Print the second last element from the 2nd dim using negative indexing
```

Out[38]:

```
5
```

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

NumPy Array Slicing

Reference: [https://www.youtube.com/watch?v=fIXh-gmR7mQ (https://www.youtube.com/watch?v=fIXh-gmR7mQ)](https://www.youtube.com/watch?v=fIXh-gmR7mQ)

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 [39]:

```
arr=np.arange(1,11)
arr
```

Out[39]:

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

In [40]:

```
# Slice elements from 1st to 5th element from array arr:
```

Out[40]:

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

Note: The result includes the start index, but excludes the end index.

In [41]:

```
# Slice elements from index 5 to the end of the array arr:
```

Out[41]:

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

In [42]:

```
# Slice elements from the beginning to index 5 (not included) in array arr:
```

Out[42]:

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

Have you heared about **Negative Slicing?**

We can use the minus operator to refer to an index from the end:

In [43]:

```
# Slice from the index 3 from the end to index 1 from the end:
```

Out[43]:

```
array([8, 9])
```

STEP

Use the step value to determine the step of the slicing:

In [44]:

```
# Print every other element from index 1 to index 7:
```

Out[44]:

```
array([2, 4, 6])
```

Did you see? using step you were able to get alternate elements within specified index numbers.

In [45]:

```
# Return every other element from the entire array arr:
```

Out[45]:

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

well done!

Lets do some slicing on 2-D array also. We already have 'a' as our 2-D array. We will use it here.

2. Array slicing in 2-D array.

In [46]:

```
# Print array a
a
```

Out[46]:

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

In [47]:

```
# From the third element, slice elements from index 1 to index 5 (not included) from array
```

Out[47]:

```
array([8, 9])
```

In [48]:

```
# In array 'a' print index 2 from all the rows :
```

Out[48]:

```
array([3, 6, 9])
```

In [49]:

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

Out[49]:

```
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

Reference: https://www.youtube.com/watch?v=h2db8BLWyVw (https://www.youtube.com/watch?v=h2db8BLWyVw)

7 points

In [50]:

```
x1= np.arange(10)
```

In [51]:

```
# assign x2 = x1
```

In [52]:

```
#print x1 and 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 [53]:

```
# change 1st element of x2 as 10
```

In [54]:

```
#Again print x1 and x2
```

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

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.

Refer: https://numpy.org/doc/stable/reference/generated/numpy.shares_memory.html (https://numpy.org/doc/stable/reference/generated/numpy.shares_memory.html)

In [55]:

```
# Check memory share between x1 and x2
```

Out[55]:

```
True
```

Hey It's True they both share memory

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

In [56]:

```
# Create a view of x1 and store it in x3.
x3 =
```

In [57]:

```
# Again check memory share between x1 and x3
```

Out[57]:

```
True
```

Woh! simple assignment is similar to view. That means 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.

Don't agree? ok lets change x3 and see if original array i.e. x1 also changes

In [58]:

```
#Change 1st element of x3=100
```

In [59]:

```
#print x1 and x3 to check if changes reflected in both
```

```
[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 [60]:

```
# Now create an array x4 which is copy of x1

x4=
```

In [61]:

```
# Change the last element of x4 as 900
```

In [62]:

```
# print both x1 and x4 to check if changes reflected in both
```

```
[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.

Still not convinced? Ok lets see if they both share memory or not

In [63]:

```
#Check memory share between x1 and x4
```

Out[63]:

```
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 origina
l 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

Reference: https://thispointer.com/python-numpy-select-elements-or-indices-by-conditions-from-numpy-array/ (https://thispointer.com/python-numpy-select-elements-or-indices-by-conditions-from-numpy-array/)

5 points

In [64]:

```
#print a
a
```

Out[64]:

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

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

In [65]:

```
# Check if every element in array a greater than 3 or not Using '>' notation
```

Out[65]:

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

In [66]:

```
# Get a list with all elements of array 'a' grater than 3
```

Out[66]:

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

In [67]:

```
# Get a list with all elements of array 'a' greater than 3 but less than 6
```

Out[67]:

```
array([4, 5])
```

In [68]:

```
# check if each elements in array 'x1' equals array 'x4' using '==' notation
```

Out[68]:

```
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

Reference: [https://www.youtube.com/watch?v=8qpMys9ptBs (https://www.youtube.com/watch?v=8qpMys9ptBs)](https://www.youtube.com/watch?v=8qpMys9ptBs)

In [69]:

```
# Print Transpose of array 'a'

#print array 'a'
print("-----------------")
```

```
[[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

Reference: [https://www.youtube.com/watch?v=p1bsYXwg97Q (https://www.youtube.com/watch?v=p1bsYXwg97Q)](https://www.youtube.com/watch?v=p1bsYXwg97Q)

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

You wanna see how? Then here we go...!

reference doc: [https://scipython.com/book/chapter-6-numpy/examples/vstack-and-hstack/ (https://scipython.com/book/chapter-6-numpy/examples/vstack-and-hstack/)](https://scipython.com/book/chapter-6-numpy/examples/vstack-and-hstack/)

In [70]:

```
# stack x1 and x4 along columns.
```

Out[70]:

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

In [71]:

```
#stack x1 and x4 along rows
```

Out[71]:

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

We hope now you saw the difference between them.

Fun fact! you can even use concatenate() function to join 2 arrays along with the axis. If axis is not explicitly passed, it is taken as 0 ie. along column

Lets try this function as well

Reference: [https://www.youtube.com/watch?v=X4zjs_wPxLU (https://www.youtube.com/watch?v=X4zjs_wPxLU)](https://www.youtube.com/watch?v=X4zjs_wPxLU)

In [72]:

```
arr1 = np.array([[1, 2], [3, 4]])

arr2 = np.array([[5, 6], [7, 8]])

##join arr1 and arr2 along rows using concatenate() function
```

Out[72]:

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

In [73]:

```
##join arr1 and arr2 along columns using concatenate() function
```

Out[73]:

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

Adding, Insert and delete Numpy array

Reference: https://www.youtube.com/watch?v=dEnCfapUbEw (https://www.youtube.com/watch?v=dEnCfapUbEw)

3 points

You can also add 2 arrays using append() function also. This function appends values to end of array

Lets see how

In [74]:

```
# append arr2 to arr1
```

Out[74]:

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

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

In [75]:

```
# Inserts values into array x1 before index 4 with elements of x4
```

Out[75]:

```
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 [76]:

```
# delete 2nd element from array x2
```

Out[76]:

```
array([100,   1,   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

Reference doc for Numpy Mathmatical functions: https://numpy.org/doc/stable/reference/routines.math.html (https://numpy.org/doc/stable/reference/routines.math.html)

8 points

In [ ]:

```
#defining a
a= np.array([[1,2,3],[4,5,6],[7,8,9]]
```

In [77]:

```
# print trigonometric sin value of each element of a
```

Out[77]:

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

In [78]:

```
# print trigonometric cos value of each element of a
```

Out[78]:

```
array([[ 0.54030231, -0.41614684, -0.9899925 ],
       [-0.65364362,  0.28366219,  0.96017029],
       [ 0.75390225, -0.14550003, -0.91113026]])
```

In [79]:

```
# Print exponential value of each elements of a
```

Out[79]:

```
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]])
```

Referal: https://numpy.org/doc/stable/reference/generated/numpy.sum.html (https://numpy.org/doc/stable/reference/generated/numpy.sum.html)

In [80]:

```
# print total sum of elements of a
```

Out[80]:

```
45
```

In [81]:

```
# Print sum in array a column wise
```

Out[81]:

```
array([ 6, 15, 24])
```

In [82]:

```
# Print sum in array a row wise
```

Out[82]:

```
array([12, 15, 18])
```

Refrence doc: https://numpy.org/doc/stable/reference/generated/numpy.median.html (https://numpy.org/doc/stable/reference/generated/numpy.median.html)

In [83]:

```
# print median of array a
```

Out[83]:

```
5.0
```

Refrence doc: https://numpy.org/doc/stable/reference/generated/numpy.std.html (https://numpy.org/doc/stable/reference/generated/numpy.std.html)

In [84]:

```
# print standard deviation of array a
```

Out[84]:

```
2.581988897471611
```

Refrence doc: https://numpy.org/doc/stable/reference/generated/numpy.linalg.det.html (https://numpy.org/doc/stable/reference/generated/numpy.linalg.det.html)

In [85]:

```
# print the determinant of array a
```

Out[85]:

```
6.66133814775094e-16
```

reference doc: https://numpy.org/doc/stable/reference/generated/numpy.linalg.inv.html (https://numpy.org/doc/stable/reference/generated/numpy.linalg.inv.html)

In [86]:

```
# print the (multiplicative) inverse of array a
```

Out[86]:

```
array([[-4.50359963e+15,  9.00719925e+15, -4.50359963e+15],
       [ 9.00719925e+15, -1.80143985e+16,  9.00719925e+15],
       [-4.50359963e+15,  9.00719925e+15, -4.50359963e+15]])
```

Reference doc: https://numpy.org/doc/stable/reference/generated/numpy.linalg.eig.html (https://numpy.org/doc/stable/reference/generated/numpy.linalg.eig.html)

In [87]:

```
# Print the eigenvalues and right eigenvectors of array a.
```

Out[87]:

```
(array([ 1.61168440e+01, -1.11684397e+00, -1.30367773e-15]),
 array([[-0.23197069, -0.78583024,  0.40824829],
        [-0.52532209, -0.08675134, -0.81649658],
        [-0.8186735 ,  0.61232756,  0.40824829]]))
```

Reference doc: https://numpy.org/doc/stable/reference/generated/numpy.dot.html (https://numpy.org/doc/stable/reference/generated/numpy.dot.html)

In [88]:

```
# compute dot product of arr1 and arr2
```

Out[88]:

```
array([[19, 22],
       [43, 50]])
```

reference doc: https://numpy.org/doc/stable/reference/generated/numpy.ndarray.max.html (https://numpy.org/doc/stable/reference/generated/numpy.ndarray.max.html)

In [89]:

```
#print largest element present in array a
```

Out[89]:

```
9
```

Reference doc: https://numpy.org/doc/stable/reference/generated/numpy.argmax.html (https://numpy.org/doc/stable/reference/generated/numpy.argmax.html)

In [90]:

```
#print index of largest element present in array a
```

Out[90]:

```
8
```

Reference doc: https://numpy.org/doc/stable/reference/generated/numpy.sort.html (https://numpy.org/doc/stable/reference/generated/numpy.sort.html)

In [91]:

```
# print sorted x4 array
```

Out[91]:

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

Reference doc: https://numpy.org/doc/stable/reference/generated/numpy.argsort.html (https://numpy.org/doc/stable/reference/generated/numpy.argsort.html)

In [92]:

```
# print indices of each sorted element in x4 array
```

Out[92]:

```
array([1, 2, 3, 4, 5, 6, 7, 8, 0, 9], dtype=int64)
```

Searching Arrays

Reference: https://www.youtube.com/watch?v=0t6FRh0Pmtw (https://www.youtube.com/watch?v=0t6FRh0Pmtw)

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 [93]:

```
# print the indexes where the value is 4 in array x1
```

Out[93]:

```
(array([4], dtype=int64),)
```

Which means that the value 4 is present at index 4

You can check it by printing array x1

In [94]:

```
#print array x1
```

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

In [95]:

```
# Print the indexes where the values are even in array x1
```

Out[95]:

```
(array([0, 2, 4, 6, 8], dtype=int64),)
```

In [96]:

```
# Print x1 where x1 is greater than 5, also if number is less than 5 then replace it with 0
```

Out[96]:

```
array([100,   0,   0,   0,   0,   0,   6,   7,   8,   9])
```

Good Job learner!

Bam! Congratulations You have completed your 4th milestone challenge too!

Its Feedback Time!

We hope you've enjoyed this course so far. We're committed to help you use "AI for All" course to its full potential, so that you have a great learning experience. And that's why we need your help in form of a feedback here.

Please fill this feedback form https://zfrmz.in/MtRG5oWXBdesm6rmSM7N (https://zfrmz.in/MtRG5oWXBdesm6rmSM7N)