Numpy Basics

In [1]:

```
#import numpy module with alias np

import numpy as 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

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

In [3]:

```
# print output
arr
```

Out[3]:

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

Dimensions in Arrays

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

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]]])

d
```

Out[4]:

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

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

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
print(np.ndim(a))
print(np.ndim(b))
print(np.ndim(c))
print(np.ndim(d))
```

```
0
1
2
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
print(a.shape)
print(b.shape)
print(c.shape)
print(d.shape)
```

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

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
print(type(a))
print(type(b))
```

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

In [9]:

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

```
3
```

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

In [ ]:

```

```

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 = [10,20,30]
l2 = [40,50,60]
l3 = np.array([l1])
l4 = np.array([l2])

l4
```

Out[10]:

```
array([[40, 50, 60]])
```

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
import timeit

mysetup = "from math import prod"

mycode = '''
l1 = [10,20,30]
l2 = [40,50,60]
Result = []
for i1,i2 in zip(l1,l2):
  Result.append(i1*i2)

print("Product is :" , Result)
'''

print (timeit.timeit(setup = mysetup,stmt = mycode,number = 10))
```

```
Product is : [400, 1000, 1800]
Product is : [400, 1000, 1800]
Product is : [400, 1000, 1800]
Product is : [400, 1000, 1800]
Product is : [400, 1000, 1800]
Product is : [400, 1000, 1800]
Product is : [400, 1000, 1800]
Product is : [400, 1000, 1800]
Product is : [400, 1000, 1800]
Product is : [400, 1000, 1800]
0.0007280000000093878
```

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]:

```
import timeit

mysetup = "from math import prod"

mycode = '''

import numpy as np
l1 = [10,20,30]
l2 = [40,50,60]
l3 = np.array([l1])
l4 = np.array([l2])
mul = np.multiply(l3,l4)

print(mul)
'''

print(timeit.timeit(setup = mysetup,stmt = mycode, number=1))
```

```
[[ 400 1000 1800]]
0.0006069000000081815
```

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.

In [13]:

```
#Create a numpy array using arange with 1 and 11 as parameter in it
print("A\n",np.arange(1,11))
```

```
A
 [ 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 parameter in iter

A = np.arange(start=1,stop=11,step=2)
print(A)
```

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

In [15]:

```
# create numpy array using eye function with 3 as passed parameter
D = np.eye(3)
D
```

Out[15]:

```
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
```

In [16]:

```
# Using arange() to generate numpy array x with numbers between 1 to 16
x=np.arange(1,17)
x
```

Out[16]:

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

In [17]:

```
# Reshape x with 2 rows and 8 columns
newx = x.reshape(2,8)
newx
```

Out[17]:

```
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 [18]:

```
# reshaoe x with dimension that will have 2 arrays that contains 4 arrays, each with 2 e
newxx = x.reshape(2,4,2)
newxx
```

Out[18]:

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

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

In [19]:

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


arr = x.reshape(-1,4)
arr
```

Out[19]:

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

In [20]:

```
# Use unknown dimention to  reshape x into 3-D numpy array with 2 arrays that contains 4
y= x.reshape(2,4,-1)


# print y
y
```

Out[20]:

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

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

In [21]:

```
# Flattening y
z = y.flatten('C')
z = y.flatten('F')
z
```

Out[21]:

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

In [22]:

```
# Create an array a with all even numbers between 1 to 17
a = np.arange(1,17)
a
bb = (a%2 ==0)
c = a[bb]


# print a
c
```

Out[22]:

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

In [23]:

```
# Get third element in array a
print(c[2])
```

```
6
```

In [24]:

```
#Print 3rd, 5th, and 7th element in array a
print(c[2:7:2])
```

```
[ 6 10 14]
```

Lets check the same for 2 D array

In [25]:

```
# 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]])
```

In [26]:

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

print(a[2,2])
```

```
9
```

Well done!

Now lets check indexing for 3 D array

In [27]:

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

In [28]:

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


print(b[1,0,2])
```

```
9
```

In [29]:

```
# Create 1D array

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

In [30]:

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

arr[0:4]
```

Out[30]:

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

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

In [31]:

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

Out[31]:

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

In [32]:

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

arr[:4]
```

Out[32]:

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

STEP

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

In [33]:

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


print(arr[1:7:2])
```

```
[2 4 6]
```

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

In [34]:

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


print(arr[: : 2])
```

```
[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.

Array slicing in 2-D array.

In [35]:

```
# Print array a
a = np.array([[1,2,3],[4,5,6]])
a
```

Out[35]:

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

In [36]:

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


print(a[1,1:5])
```

```
[5 6]
```

In [37]:

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

print(a[0:2,2])
```

```
[3 6]
```

In [38]:

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


print(a[1:])
```

```
[[4 5 6]]
```

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

Numpy copy vs view

In [39]:

```
x1= np.array([2,4,6,8])
```

In [41]:

```
# assign x2 = x1

x2 = x1
```

In [42]:

```
#print x1 and x2

print(x1)
print(x2)
```

```
[2 4 6 8]
[2 4 6 8]
```

Ok now you have seen that both of them are same

In [43]:

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


x2[0] = 10
```

In [45]:

```
#Again print x1 and x2
print(x1)
print(x2)
```

```
[10  4  6  8]
[10  4  6  8]
```

In [46]:

```
# Check memory share between x1 and x2

print("Memory size of numpy array in bytes:",
      x1.size * x1.itemsize)
print("Memory size of numpy array in bytes:",
      x2.size * x2.itemsize)
```

```
Memory size of numpy array in bytes: 16
Memory size of numpy array in bytes: 16
```

Hey It's True they both share memory

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

In [47]:

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

In [48]:

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

print("Memory size of numpy array in bytes:",
      x1.nbytes)
print("Memory size of numpy array in bytes:",
      x3.nbytes)
```

```
Memory size of numpy array in bytes: 16
Memory size of numpy array in bytes: 16
```

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 [49]:

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

x3[0] = 100
```

In [50]:

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

```
[100   4   6   8]
[100   4   6   8]
```

Now its proved.

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

In [51]:

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

x4 = x1.copy()
```

In [52]:

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

x4[3] = 900
```

In [53]:

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

print(x1)
print(x4)
```

```
[100   4   6   8]
[100   4   6 900]
```

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 [54]:

```
#Check memory share between x1 and x4


print(x1.nbytes)
print(x4.nbytes)
```

```
16
16
```

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 [55]:

```
# stack x1 and x4 along columns.

res = np.vstack((x1,x4))
res
```

Out[55]:

```
array([[100,   4,   6,   8],
       [100,   4,   6, 900]])
```

In [56]:

```
#stack x1 and x4 along rows
res1 = np.hstack((x1,x4))
res1
```

Out[56]:

```
array([100,   4,   6,   8, 100,   4,   6, 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

In [57]:

```
arr1 = np.array([[1,1,1,1],[1,0,1,0]])
arr2 = np.array([[1,2,1,2],[0,1,0,1]])


##join arr1 and arr2 along rows using concatenate() function
newarr = np.concatenate((arr1,arr2),axis=0)
newarr
```

Out[57]:

```
array([[1, 1, 1, 1],
       [1, 0, 1, 0],
       [1, 2, 1, 2],
       [0, 1, 0, 1]])
```

In [58]:

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

newarr = np.concatenate((arr1,arr2),axis=1)
newarr
```

Out[58]:

```
array([[1, 1, 1, 1, 1, 2, 1, 2],
       [1, 0, 1, 0, 0, 1, 0, 1]])
```

Adding, Insert and delete Numpy array

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

Lets see how

In [59]:

```
# append arr2 to arr1
np.append(arr1,arr2)
```

Out[59]:

```
array([1, 1, 1, 1, 1, 0, 1, 0, 1, 2, 1, 2, 0, 1, 0, 1])
```

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

In [60]:

```
# Inserts values into array x1 before index 4 with elements of x4
np.insert(x1,3,x4)
```

Out[60]:

```
array([100,   4,   6, 100,   4,   6, 900,   8])
```

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

In [61]:

```
# delete 2nd element from array x2


np.delete(x2,1)
```

Out[61]:

```
array([100,   6,   8])
```

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

Good Job learner!

In [ ]:

```

```