DECLARING NUMPY ARRAY

NOTE: HAVE COVERED MOST OF THE IMPORTANT USED FUNCTIONS WHICH ARE LITTLE TRICKY

```
import numpy as np
a=np.array([5,3,'v',7,5,6])
print(a)
print(a.dtype)
```

```
['5' '3' 'v' '7' '5' '6']
<U21
```

```
#Reshaping arrays
b=a.reshape(2,3)
print(b),print(b.shape),print(b.ndim);print(b.itemsize)
```

```
[['5' '3' 'v']
 ['7' '5' '6']]
(2, 3)
2
84
```

```
#explicitly giving n dimensional structure
c=np.array([[1,2,3],[3,4,5],[3,4,6]])
c.ndim,c.size,len(c),c.itemsize# size of the int in this case-->8 bytes
```

```
(2, 9, 3, 8)
```

```
print(b.base is a)# b shares memory with a so true
print(a.base is None)# a is the parent memory so true
print(c.base is None)# c is the parent memory so true
print(c.base is b)# c does not share memory with b or it did not derive from b so false
```

```
True
True
True
False
```

SLICING TECHNIQUES

```
print('second row of c',c[1]);print('second row of c',c[1:2])
print('      ')
print('first row of c',c[:1]);print('first row of c',c[0]);print('first row of c',c[0:1])
```

```
second row of c [3 4 5]
second row of c [[3 4 5]]
```

```
first row of c [[1 2 3]]
first row of c [1 2 3]
first row of c [[1 2 3]]
```

```
# going into an element of ndim
print('second row of first element c',c[1][0]);print('second row of third c',c[1:2][0][2])
#that is the reason this is 2 d array as it is the minimal num of ways to reach the elemen
```

```
second row of first element c 3
second row of third c 5
```

```
#multi-indexing
c[[1,2],[0,2]]
```

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

```
c[np.ix_([0,1],[2,3])]# this will give combinations to index like (0,2) (0,3) (1,2) (1,3)
```

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

```
#reshaping it to 3 dim
d=c.reshape(3,1,3)
d
```

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

       [[3, 4, 5]],

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

UNDERSTANDING TRANSPOSE

```
d.transpose(0,1,2)# all the axes needed in the transpose, so it works on the order given
```

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

       [[3, 4, 5]],

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

```
d.transpose(2,0,1)
```

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

       [[2],
        [4],
        [4]],

       [[3],
```

```
        [5],
        [6]]])
```

```
d.transpose(0,2,1)
```

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

       [[3],
        [4],
        [5]],

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

```
e=d.reshape(1,3,3)
e
```

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

```
e[0,1,2]
```

```
5
```

```
np.count_nonzero(d)# this will give non zero numbers
```

```
9
```

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

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

```
np.diag(z)#diagonal
```

```
array([1, 5])
```

```
np.diag(z,-1)# this tries to give one below the main diagonal
```

```
np.diag(z,1)#this tries to give the diagonal one above the main diagonal
```

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

```
np.trace(z)#sum of diagonal
```

```
6
```

INITIALISING ARRAYS

```
a=np.zeros(8)
b=np.zeros((3,2))#giving shape will work
a,b
```

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

```
np.eye(4)# returns a 2d array with all 1's in diagonal and 0's elsewhere
```

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

```
np.ones((2,3)),np.isnan(np.ones((2,3)))# is nan evaluates any item in the array if it is n
```

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

```
a=np.array([1,2,3,'u'])
a.nbytes# sum of each element data type size
```

```
336
```

```
b=np.array([[1,4,7],[3,4,5]])
np.full_like(b,8)# this changes all the elements to 8
```

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

```
z=np.arange(8,dtype=int)
z #it gives range of numbers until n-1
```

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

```
#Note: reshape will never make save changes to the actual one. It needs to be assigned
z.reshape(2,4)
```

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

```
z
```

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

```
np.full([2,3],7)# this gives you provision of adjusting a new shape and put the number giv
```

```
array([[7, 7, 7],
       [7, 7, 7]])
```

```
np.full_like([2,3],7)# this takes [2,3] as an array and modifies it to [7,7]
```

```
array([7, 7])
```

SPECIAL FUNCTIONS

```
a=np.full([2,2],7)
np.fill_diagonal(a,9)
a# this fills only diagonal
```

```
array([[9, 7],
       [7, 9]])
```

```
a.flatten()# this becomes a one dimension, this kills higher dimension
```

```
array([9, 7, 7, 9])
```

```
b=np.arange(8,dtype=int).reshape(2,4)
b.ravel()# this also kills dimension, it flattens
```

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

```
b=np.arange(8,dtype=int).reshape(2,4)
b
```

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

```
#if you need only 2 cols and the matrix need to be adjusted accordingly for the above arra
c=b.reshape(-1,2)# i am only guaranteed about the number of cols
c
```

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

```
c==5# this will search where the element exists and returns True
```

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

```
x=np.array([[1,4],[9,5]])
x[x==9]# this returns the true value ie, 9
```

```
array([9])
```

```
np.where(x==9)
```

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

```
np.repeat(x,2,axis=0)# axis 0 is row, 2 states that each row needs to be repeated
```

```
array([[1, 4],
       [1, 4],
       [9, 5],
       [9, 5]])
```

```
np.repeat(x,3,axis=1)# axis=1 is column wise and each one repeats 3 times
```

```
array([[1, 1, 1, 4, 4, 4],
       [9, 9, 9, 5, 5, 5]])
```

```
q=np.arange(12,dtype=int).reshape(3,4)
q
```

```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

```
j=q[np.newaxis]
j
```

```
array([[[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [ 8,  9, 10, 11]]])
```

```
j.shape
```

```
(1, 3, 4)
```

```
t=q[np.newaxis,:]
t# this produces same result as above
```

```
array([[[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [ 8,  9, 10, 11]]])
```

```
k=q[:,np.newaxis]
k# new axis is getting added to where you reach your point, its because you are trying to
```

```
array([[[ 0,  1,  2,  3]],
```

```
            [[ 4,  5,  6,  7]],

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

```
k=q[:,np.newaxis,np.newaxis]
k
```

```
     array([[[[ 0,  1,  2,  3]]],


            [[[ 4,  5,  6,  7]]],


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

```
a=np.arange(10,dtype=int)
a[:,np.newaxis]# this new axis is added to the last for the one dimensional number, so eve
```

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

```
a=np.array([2,3,5])
np.vander(a,3)# observe first columns: 4, 9, 25  second columns= 2,3,5   third column=2^0,
```

```
     array([[ 4,  2,  1],
            [ 9,  3,  1],
            [25,  5,  1]])
```

```
# you can also use column stack as well
n=3
s=np.column_stack([a**(n-i-1) for i in range(n)])
s
```

```
     array([[ 4,  2,  1],
            [ 9,  3,  1],
            [25,  5,  1]])
```

```
np.vander(a,3,increasing=True)# increasing order of your column stack
```

```
     array([[ 1,  2,  4],
            [ 1,  3,  9],
            [ 1,  5, 25]])
```

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

```
np.hstack([a,b])#stacks side by side
```

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

```
c=np.array([a]+[b])
c
```

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

```
c=np.vstack([a,b])
c
```

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

```
a=np.array([[2,3,4],[4,1,7]])
b=np.array([1,2,4])
np.vstack([a,b])#vstack is not so bothered on dimensionality difference between a and b. h
```

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

```
a
```

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

```
np.any(a==4),np.all(a==3)
```

```
(True, False)
```

0s completed at 11:54 PM