NumPy Foundaton

```
(numpy.array(object, dtype = None, copy = True, order = None, ndmin = 0))
                1. object :Any object exposing the array interface method returns an array, or any (nested)
sequence.
                2. dtype  :Desired data type of array, optional
                3. copy   :(Optional). By default (true), the object is copied
                4. order  :C (row major) or F (column major) or A (any) (default)
                6. ndmin  :Specifies minimum dimensions of resultant array
```

*NumPy short for numerical python, is one of the most important foundational packages for numerical computing in python. Most computational packages providing scientific functionality uses NumPy's array.

*while NumPy by itself do not provide modeling or scientific functionality ,having an understanding of NumPy arrays and array-oriented computing will help you use tools with array oriented semantics ,like pandas ,scipy , much more effectively

*Numeric, the ancestor of NumPy, was developed by Jim Hugunin. Another package Numarray was also developed, having some additional functionalities. In 2005, Travis Oliphant created NumPy package by incorporating the features of Numarray into Numeric package. There are many contributors to this open source project.

NumPy – A Replacement for MatLab

*NumPy is often used along with packages like SciPy (Scientific Python) and Mat−plotlib (plotting library). This combination is widely used as a replacement for MatLab, a popular platform for technical computing. However, Python alternative to MatLab is now seen as a more modern and complete programming language.

```
(It is open source, which is an added advantage of NumPy.)
```

reasons why numpy is important ::

- one of the important reason why NumPy is so important is that it is designed to efficiently do numarical computation on largrs arrays of data .
- NumPy arrays uses much less memory than built-in Python sequences.
- NumPy operations perform complex computations on entire arrays without the need for python for loops.
- The most important object defined in NumPy is an N-dimensional array type called ndarray.
- Items in the collection can be accessed using a zero-based index.
- Every item in an ndarray takes the same size of block in the memory.
- Each element in ndarray is an object of data-type object (called dtype).

The Numpy ndarray : A Multidimentional Array Object

- one of the key features of NumPy is its N-dimentional array object ,or ndarray , which is a fast ,flexible container for large datasets in python .
- An nd array is a generic multidimentional container of homogenious data.
- NumPy based algorithms are 10 to 100 times faster than their pure python counterparts and use significantly less memory

lets see

In [ ]:
```
ar
```

In [29]:
```
l=list(range(10000))
import sys
print(sys.getsizeof(1)*10000)

arr = np.arange(10000)
print(arr.itemsize *arr.size)
```

```
280000
40000
```

In [2]:
```
import numpy as np
nd_arr = np.arange(1000000)  # in Nd-array np.aranges() works same as list(range()) in lists
```

In [3]:
```
nd_arr
```

Out[3]:
```
array([     0,      1,      2, ..., 999997, 999998, 999999])
```

In [4]:
```
n_lst = list(range(1000000))  # arange the numbers data in list
```

In [6]:
```
%time nd_arr *2
```

```
Wall time: 3.01 ms
```

Out[6]:
```
array([      0,       2,       4, ..., 1999994, 1999996, 1999998])
```

In [ ]:

In [7]:
```
lst =[]
%time for i in range(1000000):lst.append(i*2)
```

```
Wall time: 228 ms
```

In [ ]:

we can see that nd-array is much more faster than list

List Vs ndarray

- lists can contain any type of data (string ,float,boolean,int)
- lists are not conventional with numarical computing efficiently as list can take any type of data .
- Numpy array is designed to do numerical computation effeciently on large no of data sets .
- numpy array can be created from other sequence types in python
- most of the packages of scientific and computational functionality uses ndarray object type | ((Numpy array , ndarray or array all refers same i.e - ndarray))

List vs array computation

In [7]:
```
#lists Vs nd array computation
list_1 = [1,2,3,4,5]
list_2 =[2,4,6,8,10]
print ("list_1 + list_2")
list_1 + list_2
#print("list_1 *list_2")
#list_1 *list_2
```

```
list_1 + list_2
```

Out[7]:
```
[1, 2, 3, 4, 5, 2, 4, 6, 8, 10]
```

In [34]:
```
# generating some random data
data = np.random.randn(2,3)
data
```

Out[34]:
```
array([[-1.36539446, -0.20604163,  1.1646133 ],
       [ 1.67867364,  0.25945128,  0.31537061]])
```

In [8]:
```
data
```

Out[8]:
```
array([[ 0.62768304,  0.07359342,  0.02007287],
       [-0.33579899, -0.0674925 , -0.49079837]])
```

In [9]:
```
data * 10
```

Out[9]:
```
array([[ 6.27683036,  0.73593415,  0.20072875],
       [-3.35798993, -0.67492497, -4.90798369]])
```

In [10]:
```
data + data
```

Out[10]:
```
array([[ 1.25536607,  0.14718683,  0.04014575],
       [-0.67159799, -0.13498499, -0.98159674]])
```

important attributes of an ndarray object are:

ndarray.ndim

```
    #the number of axes (dimensions) of the array.
```

ndarray.shape

```
    #the dimensions of the array. This is a tuple of integers indicating the size of the array in e
ach dimension. For a matrix with n rows and m columns, shape will be (n,m). The length of the shape
tuple is therefore the number of axes, ndim.
```

ndarray.size

```
    #the total number of elements of the array. This is equal to the product of the elements of sha
pe.
```

ndarray.dtype

```
    #an object describing the type of the elements in the array. One can create or specify dtype’s
 using standard Python types. Additionally NumPy provides types of its own. numpy.int32, numpy.int1
6, and numpy.float64 are some examples.
```

ndarray.itemsize

```
    #the size in bytes of each element of the array. For example, an array of elements of type floa
t64 has itemsize 8 (=64/8), while one of type complex32 has itemsize 4 (=32/8). It is equivalent to
ndarray.dtype.itemsize.
```

Creating nd array

In [8]:
```
# easiest way to create array is to use array function : np.array(list)
d =[1,'i',3,4.3j]

arr1 = np.array(d)
arr1.dtype
```

Out[8]:
```
dtype('<U11')
```

In [14]:
```
arr1.dtype
```

Out[14]:
```
dtype('int32')
```

In [24]:
```
arr1.shape
```

Out[24]:
```
(4,)
```

Nested sequences ,like a list of equal -length lists ,will be converted into a multidimentional array

In [35]:
```
data2 = [[1,"b",3],[4,5,6]] # since its a list of list ,the NumPy array has two dimentions .
arr2 = np.array(data2)
arr2
```

Out[35]:
```
array([['1', 'b', '3'],
       ['4', '5', '6']], dtype='<U11')
```

we can confirm the dimentions by : nd_array.ndim or nd_array.shape functions

In [26]:
```
arr2.size # total no of elements
```
Out[26]: 2

In [18]:
```
arr2.ndim
```
Out[18]: 2

In [19]:
```
arr2.shape # no of rows and columns
```
Out[19]: (2, 3)

In [28]:
```
#arr2.reshape(2,3)
arr2.itemsize
#arr2.dtype
```
Out[28]: 8

In [36]:
```
arr2.ravel()  # flatten the array and make it one dimentional
              # it will not affect the existing array it will create a new array

#arr.flatten()
```
Out[36]:
```
array(['1', 'b', '3', '4', '5', '6'], dtype='<U11')
```

there are some functions to create new arrays

In [22]:
```
np.arange(10) #  like the built in range but returns an ndarray instead of list
```
Out[22]:
```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

np.arange(start,stop,steps)

In [40]:
```
np.zeros((2,3)) #producing array of zero's
```
Out[40]:
```
array([[0., 0., 0.],
       [0., 0., 0.]])
```

In [68]:
```
np.ones((2,3,4)) # producing array of one's
```
Out[68]:
```
array([[[ 1.,  1.,  1.,  1.],
        [ 1.,  1.,  1.,  1.],
        [ 1.,  1.,  1.,  1.]],

       [[ 1.,  1.,  1.,  1.],
        [ 1.,  1.,  1.,  1.],
        [ 1.,  1.,  1.,  1.]]])
```

In [65]:
```
np.empty(10)# creating new array by allocating new memory with random values
```
Out[65]:
```
array([ 1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.])
```

In [48]:
```
np.linspace(1,10)
```
Out[48]:
```
array([ 1.        ,  1.18367347,  1.36734694,  1.55102041,  1.73469388,
        1.91836735,  2.10204082,  2.28571429,  2.46938776,  2.65306122,
        2.83673469,  3.02040816,  3.20408163,  3.3877551 ,  3.57142857,
        3.75510204,  3.93877551,  4.12244898,  4.30612245,  4.48979592,
        4.67346939,  4.85714286,  5.04081633,  5.2244898 ,  5.40816327,
        5.59183673,  5.7755102 ,  5.95918367,  6.14285714,  6.32653061,
        6.51020408,  6.69387755,  6.87755102,  7.06122449,  7.24489796,
        7.42857143,  7.6122449 ,  7.79591837,  7.97959184,  8.16326531,
        8.34693878,  8.53061224,  8.71428571,  8.89795918,  9.08163265,
        9.26530612,  9.44897959,  9.63265306,  9.81632653, 10.        ])
```

Data Types for ndarrays

In [50]:
```
arr_1 = np.array([1,2,3], dtype=float)
arr_2 = np.array([1,2,3],dtype=int)
```

In [48]:
```
arr_1.dtype
```
Out[48]: dtype('float64')

In [46]:
```
arr_2.dtype
```
Out[46]: dtype('int32')

You can explicitly convert or cast an array from one dtype to another using ndarray's function:: nd_array.astype()

In [53]:
```
float_arr =arr_2.astype(np.float64)
float_arr
```
Out[53]: array([1., 2., 3.])

In [81]:
```
float_arr.dtype
```
Out[81]: dtype('float64')

In [51]:
```
arr_3 =np.array([2.1,3.2,2.4,4.2])
```

In [52]:
```
arr_3
```
Out[52]: array([2.1, 3.2, 2.4, 4.2])

In [54]:
```
arr_31 =arr_3.astype(int)
```

In [55]:
```
arr_31.dtype
```
Out[55]: dtype('int32')

In [86]:
```
arr_31
```
Out[86]: array([2, 3, 2, 4])

In [21]:
```
x = np.array([[1, 2], [3, 4], [5, 6]])
y = x[[0,1,2], [0,1,0]]
print(y)
```
```
[1 4 5]
```

Arithmetic with NumPy Arrays

In [87]:
```
arr_1 +arr_2
```
Out[87]: array([2., 4., 6.])

In [88]:
```
arr_1 * arr_2
```
Out[88]: array([1., 4., 9.])

In [89]:
```
arr_1/2
```
Out[89]: array([0.5, 1. , 1.5])

In [90]:
```
arr_2-arr_1
```
Out[90]: array([0., 0., 0.])

In [91]:
```
arr_1**2
```
Out[91]: array([1., 4., 9.])

In [92]:
```
1/arr_1
```
Out[92]: array([1.   , 0.5   , 0.33333333])

In [93]:
```
arr_2<2
```
Out[93]:
```
array([ True, False, False], dtype=bool)
```
In [94]:
```
arr_1==arr_2
```
Out[94]:
```
array([ True,  True,  True], dtype=bool)
```

find BMI : B = weight(kg)/ height**2(m)

In [28]:
```
student=['A','B','C','D','E','F','G']
w=[49.0,51.0,54.0,60.0,85.0,75.0,90.0]
#w=map(float(),range(w))
h=[1.54,1.60,1.59,1.64,1.78,1.76,1.80]
#BMI= weight/height**2
print(w)
```
```
[49.0, 51.0, 54.0, 60.0, 85.0, 75.0, 90.0]
```
In [34]:
```
for i,j in zip(w,h):
    bmi=i/(j*j)
    print("BMI::>" , bmi)
```
```
BMI::> 20.66115702479339
BMI::> 19.921874999999996
BMI::> 21.359914560341757
BMI::> 22.308149910767405
BMI::> 26.82742078020452
BMI::> 24.212293388429753
BMI::> 27.777777777777775
```
In [116]:
```
bmi=[x/y**2 for x,y in zip(w,h)]
bmi
```
Out[116]:
```
[20.66115702479339,
 19.921874999999996,
 21.359914560341757,
 22.308149910767405,
 26.82742078020452,
 24.212293388429753,
 27.777777777777775]
```
In [40]:
```
w_arr= np.array(w)
h_arr=np.array(h)
print(type(w_arr),h_arr)
```
```
<class 'numpy.ndarray'> [1.54 1.6  1.59 1.64 1.78 1.76 1.8 ]
```
In [42]:
```
bmi= w_arr/(h_arr*2)
bmi
```
Out[42]:
```
array([15.90909091, 15.9375    , 16.98113208, 18.29268293, 23.87640449,
       21.30681818, 25.        ])
```
In [71]:
```
np.sum(np.arange(10))
```
Out[71]:
```
45
```
In [98]:
```
np.short()
```
Out[98]:
```
0
```

mathematical functions an array can perform

In [143]:
```
a = np.array([2,3,4,5,6,7,8,9,10])
a.min()
a.max()
a.sum()
a.mean()
np.sqrt(a)
np.exp(a)
np.log(a)
np.std(a)
```
Out[143]:
```
2.581988897471611
```
In [146]:
```
# can do a matrix product
a.dot(b)
```
Out[146]:
```
array([ 3.6,  5.4,  7.2,  9. , 10.8, 12.6, 14.4, 16.2, 18. ])
```
Indexing and Slicing

In [ ]:

In [75]:
```
arr_3=np.arange(10)
arr_3>2
```
Out[75]:
```
array([False, False, False,  True,  True,  True,  True,  True,  True,  True], dtype=bool)
```
In [104]:
```
arr_old=arr_1[1:2].copy()
```
In [105]:
```
arr_old
```
Out[105]:
```
array([ 2.])
```
In [15]:
```
arr_3d= np.array([[[1,2,3],[4,5,6]],[[3,4,5],[6,7,8]]])
```
In [16]:
```
arr_3d
```
Out[16]:
```
array([[[1, 2, 3],
        [4, 5, 6]],

       [[3, 4, 5],
        [6, 7, 8]]])
```
In [17]:
```
arr_3d.shape
```
Out[17]:
```
(2, 2, 3)
```
In [18]:
```
arr_3d
```
Out[18]:
```
array([[[1, 2, 3],
        [4, 5, 6]],

       [[3, 4, 5],
        [6, 7, 8]]])
```
In [19]:
```
arr_3d[1,0]=322
```
In [20]:
```
arr_3d
```
Out[20]:
```
array([[[  1,   2,   3],
        [  4,   5,   6]],

       [[322, 322, 322],
        [  6,   7,   8]]])
```
In [21]:
```
pd =arr_3d
```
In [22]:
```
import numpy as np
```
Index slicing

In [23]:
```
x_1=np.array([[1,2,3],[4,5,6],[2,1,3]])
```

In [24]:
```
x_1
```

Out[24]:
```
array([[1, 2, 3],
       [4, 5, 6],
       [2, 1, 3]])
```

In [25]:
```
x_1[0][0]
```

Out[25]:
```
1
```

in multi-dimentional array ,if you omit later indices, it will return a lower dimention array ,consisting of all the data of higher dimentions

In [26]:
```
arr_3d
```

Out[26]:
```
array([[[  1,   2,   3],
        [  4,   5,   6]],

       [[322, 322, 322],
        [  6,   7,   8]]])
```

In [77]:
```
dpd=arr_3d[0]
dpd.ndim
```

Out[77]:
```
2
```

In [28]:
```
arr_3d.ndim
```

Out[28]:
```
3
```

In [29]:
```
oldvalues=arr_3d[0].copy
```

slicing

In [33]:
```
arr_3d[:2]=0
```

In [34]:
```
arr_3d
```

Out[34]:
```
array([[[0, 0, 0],
        [0, 0, 0]],

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

you can pass multiple slices just like you can pass multiple indexes.

In [68]:
```
arr_2nd=np.array([[1,2,3],[2,3,4],[6,7,8]])
arr_2nd
```

Out[68]:
```
array([[1, 2, 3],
       [2, 3, 4],
       [6, 7, 8]])
```

In [69]:
```
arr_slic = arr_2nd[:1,:2].copy()
arr_slic
arr_2nd[:1,:2]=22
arr_2nd
```

Out[69]:
```
array([[22, 22,  3],
       [ 2,  3,  4],
       [ 6,  7,  8]])
```

In [71]:
```
arr_2nd[:1,:2]=arr_slic
arr_2nd
```

Out[71]:
```
array([[1, 2, 3],
       [2, 3, 4],
       [6, 7, 8]])
```

In [23]:
```
lst_1=[1,2,3,5,5,6,7]
lst_slic=lst_1[2:5]
lst_slic
```
Out[23]:
```
[3, 5, 5]
```
In [24]:
```
lst_slic[2]=100
lst_slic
```
Out[24]:
```
[3, 5, 100]
```
In [25]:
```
lst_1
```
Out[25]:
```
[1, 2, 3, 5, 5, 6, 7]
```

stacking

In [11]:
```
aa=np.arange(6).reshape(3,2)
bb=np.arange(6,12).reshape(3,2)
np.vstack((aa,bb))      # vertical stacking argyment would be a tupple of arrays
#np.hstack((aa,bb))     # horizontal stacking
```
Out[11]:
```
array([[ 0,  1],
       [ 2,  3],
       [ 4,  5],
       [ 6,  7],
       [ 8,  9],
       [10, 11]])
```

Split array

In [19]:
```
ab=np.arange(30).reshape(3,10)

# can split the array into equal sized  arrays (virtically or horizontally)

ab
#np.vsplit(ab,2)
np.hsplit(ab,2)
```
Out[19]:
```
[array([[ 0,  1,  2,  3,  4],
        [10, 11, 12, 13, 14],
        [20, 21, 22, 23, 24]]), array([[ 5,  6,  7,  8,  9],
        [15, 16, 17, 18, 19],
        [25, 26, 27, 28, 29]])]
```
In [ ]:
```
np.transpose(my_array)
my_array.flatten()
np.concatenate((array_1, array_2, array_3))

np.poly()      # we pass the roots to get the polinomial
np.roots()     # we can find the roots by passing the coeficients
np.polyint()   # integral
np.polyder()   # derivative
np.polyval(p,x) # evaluates a polynomial p at value x

np.linalg.det([[1 , 2], [2, 1]]) # calculates the determinant of square matrics

vals, vecs = np.linalg.eig([[1 , 2], [2, 1]]) # calculates eigen value and vectors

np.linalg.inv([[1 , 2], [2, 1]]) # calculates multiplicative inverse
```
In [ ]:
```

```