What is numpy?

NumPy is the fundamental package for scientific computing in Python. It is a Python library that provides a multidimensional array object, various derived objects (such as masked arrays and matrices), and an assortment of routines for fast operations on arrays, including mathematical, logical, shape manipulation, sorting, selecting, I/O, discrete Fourier transforms, basic linear algebra, basic statistical operations, random simulation and much more.

At the core of the NumPy package, is the ndarray object. This encapsulates n-dimensional arrays of homogeneous data types

Numpy Arrays Vs Python Sequences

- NumPy arrays have a fixed size at creation, unlike Python lists (which can grow dynamically). Changing the size of an ndarray will create a new array and delete the original.
- The elements in a NumPy array are all required to be of the same data type, and thus will be the same size in memory.
- NumPy arrays facilitate advanced mathematical and other types of operations on large numbers of data. Typically, such operations are executed more efficiently and with less code than is possible using Python's built-in sequences.
- A growing plethora of scientific and mathematical Python-based packages are using NumPy arrays; though these typically support Python-sequence input, they convert such input to NumPy arrays prior to processing, and they often output NumPy arrays.

In [4]:
```
import numpy as np
```

Creating Numpy Arrays

In [1]:
```
# np.array
# 1d          --- we can call it vector
import numpy as np
a = np.array([1,2,3,4,5])
a
```

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

In [2]:
```
# 2d  ---can call it matrix
import numpy as np
b = np.array([[1,2,3],[4,5,6]])
b
```

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

In [4]:
```
# 3d  ----- can call it tensor
c = np.array([[[1,2],[3,4],[5,6]]])
c
```

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

In [7]:
```
# dtype
np.array([1,2,3,4], dtype='int')
```

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

In [10]:
```
np.array([1,2,3,4,5], dtype='float')
```

Out[10]:
```
array([1., 2., 3., 4., 5.])
```

In [11]:
```
np.array([1,2,3,4,5], dtype='complex')
```

Out[11]:
```
array([1.+0.j, 2.+0.j, 3.+0.j, 4.+0.j, 5.+0.j])
```

In [14]:
```
# np.arrange
np.arange(1,11,1)   #start-> end(excluding)-> difference
```

Out[14]:
```
array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10])
```

In [5]:
```
# with reshape
np.arange(1,11).reshape(5,2)
```

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

In [6]:
```
np.arange(1,11).reshape(2,5)
```

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

In [7]:
```
# np.zeros  - creating 0s
np.zeros(10)
```

Out[7]:
```
array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])
```

In [9]:
```
# np.ones  - creating 1s
np.ones((3,4))  # 3 rows, 4 columns
```

Out[9]:
```
array([[1., 1., 1., 1.],
       [1., 1., 1., 1.],
       [1., 1., 1., 1.]])
```

In [12]:
```
# np.random
np.random.random(10)
```

Out[12]:
```
array([0.0502971 , 0.78001039, 0.2208413 , 0.11209118, 0.28595491,
       0.84801468, 0.61709181, 0.64291714, 0.05756206, 0.6953999 ])
```

In [16]:
```
# np.linspace    --linear space --do generate linearly separated value
np.linspace(10,50,5)  # lower range -> upper range-> number of items you want
```

Out[16]:
```
array([10., 20., 30., 40., 50.])
```

In [17]:
```
# np.identity
np.identity(3)
```

Out[17]:
```
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
```

Array Attributes

In [29]:
```
a1 = np.arange(10)
a2 = np.arange(12, dtype=float).reshape(3,4)
a3 = np.arange(8).reshape(2,2,2) # (2 2d array)  -> shape(2,2))
print(a1)
print('\n')
print(a2)
print('\n')
print(a3)
```

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


[[ 0.  1.  2.  3.]
 [ 4.  5.  6.  7.]
 [ 8.  9. 10. 11.]]


[[[0 1]
  [2 3]]

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

In [31]:
```
# ndim ---tells us dimention
print(a1.ndim)
print(a2.ndim)
print(a3.ndim)
```

```
1
2
3
```

In [32]:
```
# shape --- items on per dimention
print(a1.shape)
print(a2.shape)
print(a3.shape)
```

```
(10,)
(3, 4)
(2, 2, 2)
```

In [33]:
```
# size    - number of elemets
print(a1.size)
print(a2.size)
print(a3.size)
```

```
10
12
8
```

In [34]:
```
# itemsize --- how many bites occupied
# int32- 4 bytes
# int64- 8 bytes, floats = 8 bytes

print(a1.itemsize)
print(a2.itemsize)
print(a3.itemsize)
```

```
4
8
4
```

In [36]:
```
# dtypes

a1 = np.arange(10)
a2 = np.arange(12, dtype=float).reshape(3,4)
a3 = np.arange(8).reshape(2,2,2)

print(a1.dtype)
print(a2.dtype)
print(a3.dtype)
```

```
int32
float64
int32
```

changing datatypes

In [41]:
```
# astype

a1 = np.arange(10)
a2 = np.arange(12, dtype=float).reshape(3,4)
a3 = np.arange(8).reshape(2,2,2)

a3.astype(np.int32)
```

Out[41]:
```
array([[[0, 1],
        [2, 3]],

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

In [42]:
```
a1.astype(np.float)
```

```
C:\Users\HP\AppData\Local\Temp\ipykernel_6860\494888521.py:1: DeprecationWarn
ing: `np.float` is a deprecated alias for the builtin `float`. To silence thi
s warning, use `float` by itself. Doing this will not modify any behavior and
is safe. If you specifically wanted the numpy scalar type, use `np.float64` h
ere.
Deprecated in NumPy 1.20; for more details and guidance: https://numpy.org/de
vdocs/release/1.20.0-notes.html#deprecations (https://numpy.org/devdocs/relea
se/1.20.0-notes.html#deprecations)
  a1.astype(np.float)
```

Out[42]:
```
array([0., 1., 2., 3., 4., 5., 6., 7., 8., 9.])
```

In [43]:
```
a2.astype(np.int32)
```

Out[43]:
```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

Array operation

In [45]:
```
import numpy as np
a1 = np.arange(12).reshape(3,4)     # 1-11
a2 = np.arange(12,24).reshape(3,4)  # 12-23
print(a1)
print('\n')
print(a2)
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]


[[12 13 14 15]
 [16 17 18 19]
 [20 21 22 23]]
```

In [47]:
```
# scaler operations
# arithmetic

print(a1*3)
print('\n')

print(a2*2)
```

```
[[ 0  3  6  9]
 [12 15 18 21]
 [24 27 30 33]]


[[24 26 28 30]
 [32 34 36 38]
 [40 42 44 46]]
```

In [48]:
```
# relational
# here doing itemwise operation
a2 > 15
```

Out[48]:
```
array([[False, False, False, False],
       [ True,  True,  True,  True],
       [ True,  True,  True,  True]])
```

In [50]:
```
# vector operation
# arithmetic

print(a1+a2)
print('\n')
print(a2-a1)
```

```
[[12 14 16 18]
 [20 22 24 26]
 [28 30 32 34]]


[[12 12 12 12]
 [12 12 12 12]
 [12 12 12 12]]
```

Array Function

In [52]:
```
a = np.random.random((3,3))
b = np.round(a*100)
print(b)
```

```
[[64. 23. 13.]
 [ 3. 10.  3.]
 [80. 57. 40.]]
```

In [57]:
```
# max/min/sum/prod

from numpy.core.fromnumeric import prod
print(np.prod(b))
print(np.min(b))
print(np.sum(b))
print(np.max(b, axis=0))  # In numpy, columnwise = 0, rowwise = 1. Here column
```

```
314136576000.0
3.0
293.0
[80. 57. 40.]
```

In [66]:
```
# mean/median/std/var
print(np.mean(b))
print(np.mean(b,axis=1))   # row wiae mean
print(np.mean(b,axis=0))   # column wiae mean
print(np.std(b))
print(np.var(b))
```

```
32.55555555555556
[33.33333333  5.33333333 59.        ]
[49.         30.         18.66666667]
27.125679146074894
735.8024691358024
```

In [69]:
```
# trigonometric function
print(np.sin(b))
print('\n')
print(np.cos(b))
```

```
[[ 0.92002604 -0.8462204   0.42016704]
 [ 0.14112001 -0.54402111  0.14112001]
 [-0.99388865  0.43616476  0.74511316]]


[[ 0.39185723 -0.53283302  0.90744678]
 [-0.9899925  -0.83907153 -0.9899925 ]
 [-0.11038724  0.89986683 -0.66693806]]
```

In [75]:
```
# dot product
# if the size is same. a(col)=b(row) or  b(row)=a(col)

a = np.arange(12).reshape(3,4)
b = np.arange(12,24).reshape(4,3)

print(np.dot(a,b))
# print(a)
# print(b)
```

```
[[114 120 126]
 [378 400 422]
 [642 680 718]]
```

In [81]:
```
# log and exponent

print(np.log(b))
print('\n')
print(np.exp(a))
```

```
[[2.48490665 2.56494936 2.63905733]
 [2.7080502  2.77258872 2.83321334]
 [2.89037176 2.94443898 2.99573227]
 [3.04452244 3.09104245 3.13549422]]


[[1.00000000e+00 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 2.20264658e+04 5.98741417e+04]]
```

In [86]:
```
# round/floor/ceil

a = np.random.random((2,3))*100
print(np.round(a))
print('\n')
print(np.ceil(a))
print('\n')
print(np.floor(a))
```

```
[[ 6. 17. 39.]
 [87.  4. 37.]]


[[ 6. 18. 40.]
 [87.  5. 37.]]


[[ 5. 17. 39.]
 [86.  4. 36.]]
```

indexing and Slicing

In [110]:
```
import numpy as np

a = np.arange(10)
b = np.arange(12).reshape(3,4)
c = np.arange(8).reshape(2,2,2)  # 2 2D array (2 row, 2 col)

print(b)
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
```

In [95]:
```


# lets find out 6 from b        -> row:1, col:2
print(b[1,2])
# lets find out 7 from b        -> row:1, col:3
print(b[1,3])
# lets find out 10 from b        -> row:2, col:2
print(b[2,2])
# lets find out 8 from b        -> row:2, col:0
print(b[2,0])
```

```
6
7
10
8
```

In [99]:
```
print(c)
```

```
[[[0 1]
  [2 3]]

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

In [111]:
```
# which part of 2d array?
# if first part then 0, if 2nd part then 1. rest is same like 2d array.

# find 7          -> 2st part of the array(index=1), row=1, col=1
print(c[1,1,1])
```

```
7
```

In [112]:
```
# lets find out 5 from c   -> 2nd part of the array(index=1), row=0, col=1
print(c[1, 0, 1])
```

```
5
```

In [113]:
```
# find 1          -> 1st part of the array(index=0), row=0, col=1
print(c[0, 0, 1])
```

```
1
```

In [114]:
```
# find 6          -> 2st part of the array(index=1), row=1, col=0
print(c[1, 1, 0])
```

```
6
```

Slicing

In [115]:
```
import numpy as np
a = np.arange(10)     # vector(1d array)
b = np.arange(12).reshape(3,4)      # matrix(2d array)
c = np.arange(8).reshape(2,2,2)  # 2 2D array (2 row, 2 col)        # Tensor(3
```

In [116]:
```
print(a)
```

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

In [117]:
```
# let's find 3 to 7 from a. first index including, last index excluding. just
print(a[3:8])
```

```
[3 4 5 6 7]
```

In [119]:
```
# let's find 3 to 7, with interval 2
print(a[3:8:2])   # first index including, second index excluding, last one is
```

```
[3 5 7]
```

In [126]:
```
print(b)
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
```

In [132]:
```
# lets find out  0 to 3
print(b[0, :4])     # 1st par is row, 2nd part is col. row:0, col:all
```

```
[0 1 2 3]
```

In [130]:
```
# lets find out 3rd column
print(b[:, 2])
```

```
[ 2  6 10]
```

In [134]:
```
# lets find out 5,6 and 9,10
print(b[1:3 , 1:3])
# print(b[1: , 1:3])
```

```
[[ 5  6]
 [ 9 10]]
```

In [138]:
```
# lets find out 0,3 and 8,11
print(b[0:3:2, 0:4:3])
# print(b[::2, ::3])
```

```
[[ 0  3]
 [ 8 11]]
[[ 0  3]
 [ 8 11]]
```

In [139]:
```
print(b)
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
```

In [142]:
```
# lets find out 1,3 and 9,11
print(b[::2, 1::2])
# print(b[0:3:2, 1:4:2])
```

```
[[ 1  3]
 [ 9 11]]
[[ 1  3]
 [ 9 11]]
```

In [143]:
```
# lets find out 4,7
print(b[1, ::3])
```

```
[4 7]
```

In [145]:
```
# lets find out 1,2,3 and 5,6,7
print(b[:2, 1:])
```

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

In [149]:
```
import numpy as np
a = np.arange(27).reshape(3,3,3)
print(a) # a3 consist of 3-2d array
```

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

 [[ 9 10 11]
  [12 13 14]
  [15 16 17]]

 [[18 19 20]
  [21 22 23]
  [24 25 26]]]
```

In [152]:
```
# let's find out
#     [[ 9, 10, 11],
#      [12, 13, 14],
#      [15, 16, 17]],

print(a[1])
```

```
[[ 9 10 11]
 [12 13 14]
 [15 16 17]]
```

In [153]:
```
# let's find out
# [[ 0,  1,  2],
# [ 3,  4,  5],
# [ 6,  7,  8]], and

#       [[18, 19, 20],
#       [21, 22, 23],
#       [24, 25, 26]]

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

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

 [[18 19 20]
  [21 22 23]
  [24 25 26]]]
```

In [156]:
```
# let's find out 2nd row of first 2d array
print(a[0, 1])
```

```
[3 4 5]
```

In [158]:
```
# let's find out 2nd column of second 2d array
print(a[1, : ,1])
```

```
[10 13 16]
```

In [159]:
```
# let's find out 22,23,25,26
print(a[2, 1:, 1:])
```

```
[[22 23]
 [25 26]]
```

In [161]:
```

print(a)
```

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

 [[ 9 10 11]
  [12 13 14]
  [15 16 17]]

 [[18 19 20]
  [21 22 23]
  [24 25 26]]]
```

In [183]:
```
# let's find out 0,2,18,20
print(a[::2, 0, ::2])
```

```
[[ 0  2]
 [18 20]]
```

iterating

In [184]:
```
import numpy as np
a = np.arange(10)
b = np.arange(12).reshape(3,4)
c = np.arange(27).reshape(3,3,3)
```

In [185]:
```
# iterate 1D array
for i in a:
    print(i)
```

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

In [186]:
```
# iterating 2d array of
for i in b:
    print(i)
```

```
[0 1 2 3]
[4 5 6 7]
[ 8  9 10 11]
```

In [189]:
```
# iterating 2d array of c
for i in c:
    print(i)
```

```
[[0 1 2]
 [3 4 5]
 [6 7 8]]
[[ 9 10 11]
 [12 13 14]
 [15 16 17]]
[[18 19 20]
 [21 22 23]
 [24 25 26]]
```

In [191]:
```
# printing indivisual numbers from 2d or 3d array
for i in np.nditer(c):
    print(i)
```

```
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
```

note:

- when we need a perticular row, we need all column.
- when we need a perticular column, we need all row.

Reshape

- reshape
- Transpose
- rave

In [192]:
```
print(b)
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
```

Transpose

convert row into column and column into row

In [196]:
```
print(np.transpose(b))
print('\n')
print(b.T)
print('\n')
print(b.transpose())
```

```
[[ 0  4  8]
 [ 1  5  9]
 [ 2  6 10]
 [ 3  7 11]]


[[ 0  4  8]
 [ 1  5  9]
 [ 2  6 10]
 [ 3  7 11]]


[[ 0  4  8]
 [ 1  5  9]
 [ 2  6 10]
 [ 3  7 11]]
```

ravel

convert any dimention into 1d array(vector)

In [200]:
```
print(b)    # 2D array(matrix)
print('\n')
print(b.ravel())
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]


[ 0  1  2  3  4  5  6  7  8  9 10 11]
```

reshape

In [204]:
```
print(b)
print('\n')
print(b.reshape(4,3))
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]


[[ 0  1  2]
 [ 3  4  5]
 [ 6  7  8]
 [ 9 10 11]]
```

Staking

stack() is used for joining multiple NumPy arrays. Unlike, concatenate(), it joins arrays along a new axis. It returns a NumPy array.

In [205]:
```
a = np.arange(12).reshape(3,4)
b = np.arange(12, 24).reshape(3,4)
```

In [206]:
```
# horizontal stacking
np.hstack((a, b))
```

Out[206]:
```
array([[ 0,  1,  2,  3, 12, 13, 14, 15],
       [ 4,  5,  6,  7, 16, 17, 18, 19],
       [ 8,  9, 10, 11, 20, 21, 22, 23]])
```

In [209]:
```
# vertical stacking
np.vstack((a,b))
```

Out[209]:
```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15],
       [16, 17, 18, 19],
       [20, 21, 22, 23]])
```

Splitting

cutting should be equal position

In [210]:
```
# horizontally cutting 2 part
np.hsplit(a, 2)
```

Out[210]:
```
[array([[0, 1],
        [4, 5],
        [8, 9]]),
 array([[ 2,  3],
        [ 6,  7],
        [10, 11]])]
```

In [211]:
```
# horizontally cutting 4 part
np.hsplit(a, 4)
```

Out[211]:
```
[array([[0],
        [4],
        [8]]),
 array([[1],
        [5],
        [9]]),
 array([[ 2],
        [ 6],
        [10]]),
 array([[ 3],
        [ 7],
        [11]])]
```

In [212]:
```
# vertically cutting 3 part
np.vsplit(a, 3)
```

Out[212]:
```
[array([[0, 1, 2, 3]]), array([[4, 5, 6, 7]]), array([[ 8,  9, 10, 11]])]
```

In [ ]:
```

```