Aug-1

Python tutorial 15

<i2 - int16
<i8 - int64
<i4 - int32

- Numpy: (Numeric Python, a library in Python)
  - Numpy array:

```
import numpy as np
np.array(["MADHU", 2, 3])
array(['MADHU', '2', '3'])
type(np.array(["MADHU", 2, 3]))
numpy.ndarray   (nd = n-dimensional)
```

  - upcasting: (like, float > int)

```
np.array([1, 2, 3.0])
array([1., 2., 3.])
```

  - 2D-array:

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

  - Minimum dimension: (by default, ndmin=0)

```
np.array([1,2,3], ndmin=5)
array([[[[[1,2,3]]]]])
```

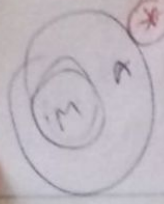
  - data type conversion: (type-casting)

```
np.array([1,2,3], dtype=complex)
array([1.+0.j, 2.+0.j, 3.+0.j])
```

  - data type consisting of more than one ele:

```
x = np.array([(1,2),(3,4)], dtype=
              [('a','<i2'),('b','<i8')])
type(x[0][1]) → numpy.int64
```

(tuples)



* matrix is sub-class of array.

np.array
np.asarray → (same return) have control over data type.

- creating array from sub-classes:

```
np.mat(np.array([[1,2],[4,7]]))
↳ matrix([[1,2],
          [4,7]])
type ↳ numpy.matrix

np.mat('1,2 ; 3,4')
↳ matrix([[1,2],
          [3,4]])
```

- numpy.asarray:

```
np.asarray((1,2))
↳ array([1,2])
```

check:

```
issubclass(np.matrix, np.ndarray)
↳ True
issubclass(np.ndarray, np.matrix)
↳ False
```

- numpy.asanyarray:

np.array :
np.asarray :
np.asanyarray :

- Creating array x, with reference y & a copy z :

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

(∵ There is no separate space for y in the memory, it just point towards x, so if we do some manipulation in x, same will happen in y & vice-versa.)

like: y[1] = 342

y → array([1, 342, 3])
x → array([1, 342, 3])

& x[0] = 932

x → array([932, 342, 3])
y → array([932, 342, 3])

(∵ Hence, we can say both x & y points to same location.) ← known as, swallow copy

→ if I want situation in which changes in x are not reflected in y & vice-versa, use copy()

```
z = np.copy(x)
```

(∵ z is pointing to different location).

z → array([932, 342, 3])

now, z[0] = 32

z → array([32, 342, 3])
x → array([932, 342, 3])

(∵ Hence, x & z points to different locations).
↑ known as, deep copy.

[max. we can visualize 3-D data.]
(we can represent but can't visualise data > 3-D)

- numpy.fromfunction: (fromfunction() takes another function to create array acc. to req.)
  - construct array by executing function over each coordinate:

```
np.fromfunction(lambda i,j: i==j,
                (3,3), dtype=int)
array([[[True, False, False],
        [False, True, False],
        [False, False, True]]])
```

(3,3) → any dimension array can be created

Note:- (4, 3, 3, 3) means we have 3x3x3 data 4-times.

- numpy.fromiter:

```
iterable = (x*x for x in range(5))
np.fromiter(iterable, float)
array([0., 1., 4., 9., 16.])
```

- numpy.fromstring:

```
a = np.fromstring('234 234', sep=' ')
a → array([234., 234.])
np.fromstring('1,2', dtype=int, sep=',')
array([1, 2])
```

{ Wherever Numpy comes into picture, means we are dealing with array }

- create record array from list of arrays:

```
x1 = np.array([1,2,3,4])
x2 = np.array(['a','dd','xyz','12'])
x3 = np.array([1.1, 2, 3, 4])
x4 = np.array([1.1, 2, 3, 4])
r = np.core.records.fromarrays
    ([x1,x2,x3,x4], names='a,b,c,d')
```

(names optional)

```
r
rec.array([[1,'a',1.1,1.1),(2,'dd',2.,2.),
           (3,'xyz',3.,3.),(4,'12',4.,4.)],
dtype=[('a','<i4'),('b','<U3'),('c','<f8'),
       ('d','<f8')])
r[1] → (2,'dd',2.,2.)
r[1]['c'] → 2.0
```

- data types:

```
my_mat = [[1,2,3],[4,5,6],[7,8,9]]
mat = np.array(my_mat)
mat.ndim → 2   (as we can represent mat in 2D)
mat.size → 9
mat.shape → (3,3)
mat.dtype → int64
```

range → consider integer data set.
arange → " int./float " "
linespace → also considers upper-bound.
(not available in python)

- arange and linspace:

list(range(4,7)) → [4,5,6]
list(range(4,10,2)) → [4,6,8].
list(range(4.4,7)) → error
(∵ floating point can't be represented as integer)

np.arange(5,15,2) → array([5,7,9,11,13])
np.arange(5,16,2.8) → array([5., 7.8, 10.6, 13.4])
np.arange(10,0,-2) → array([10,8,6,4,2])

- Linespace: (initial, end, no. of data)

(step-size = $\frac{\text{end - start}}{\text{no. of data}}$)

np.linespace(1,5,5)
array([1., 2., 3., 4., 5.])
np.linespace(1,5,10)
array([1., 1.444, 1.889, 2.333, 2.778,
3.222, 3.667, 4.111, 4.556, 5.])

Note: in range/arange we give step-size while in linespace " " no. of data.

Note: in linespace, we can also write
(axis = 0/1, endpoint = True/False,
dtype = None/int/float,
retstep = True (for finding step-size))

Q. (at last of this lecture).

- Matrix creation:

① zeros: (.zeros)

```
print(np.zeros(5)) → [0. 0. 0. 0. 0.]
print(np.zeros(2,3)) → [[0. 0. 0.]
                        [0. 0. 0.]]
```

② ones: (.ones)

```
print(np.ones(2,2,3))
[[[1. 1. 1.]
  [1. 1. 1.]]
 [[1. 1. 1.]
  [1. 1. 1.]]
 [[1. 1. 1.]
  [1. 1. 1.]]]
```

(2×3) matrix 3 2-times, with entries (1)

③ any number same in whole matrix:

```
print(np.ones(2,3))
[[6. 6. 6.]
 [6. 6. 6.]]
```

④ identity matrix: (.eye) (always sq. matrix)

```
print(np.eye(3)) → [[1. 0. 0.]
                    [0. 1. 0.]
                    [0. 0. 1.]]
```

(3×3)

- logspace: (log of data)

```
np.logspace(2.0, 3.0, num=4, base=10)
→ array([100., 215.443469, 464.15883361, 1000.])
```

start, end, number of data, by default