* NumPy Revision *

↳ Numerical Python.

↳ Deal with nums, used to perform math opera^n on 1-D array.

↳ Numpy array: Homogeneous data, same dtype to array.

- **Why NumPy:**

① Time-efficient.

② Has many math operations.

③ Probability.

④ Linear algebra.

⑤ Matrix related problems.

⑥ statistics & Logarithms.

⑦ Random fun^n (Generate random number).

● Get started:-

① **Install numpy library:** Use pip command:

```
pip install numpy
```

② **Import numpy:** use following import statement:

```
import numpy as np
```

③ **Check version:** Use --version-- attribute:

```
np.--version--
```

● Create Arrays in NumPy:-

① 1-D Array:

```
>>>arr = np.array ([1, 2, 3, 4, 5])
```

↳ Represented by <class 'numpy.ndarray'>

② 2-D Array:

↳ Array having elements as arrays.

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

O/P →
```
[[1  2  3]
 [4  5  6]]
```

③ Higher Dimensional Array:

↳ Array with high-dim array elements.

```
>>> arr = np.array ([1, 2, 3, 4], ndmin = 5)
```

O/P →
```
[[[[[1  2  3  4]]]]]
```

5 - D Array

④ Reshape: create array with diff shape.

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

>>> arr. = a. reshape (3, 3)
```

3*3 must equal to No. of elements.

O/P →
```
[[1  2  3]
 [4  5  6]
 [7  8  9]]
```

create 2D array from 1D array.

⑤ Flattening arrays:-

- Convert multi-dim array to 1D array.
- Use reshape(-1) for flattening.

```
>>> a = np.array ([[1, 2, 3], [4, 5, 6]])
>>> arr = ar. reshape (-1)
```

O/p → [1 2 3 4 5 6]

create 1D array.

⑥ convert np array to list:-

- convert 'numpy.ndarray' to 'list' data type.

```
>>> arr = np. array ([1, 2, 3, 4, 5])
    <class 'numpy.ndarray'> [1 2 3 4 5]

>>> l = arr.tolist()
    <class 'list'> [1, 2, 3, 4, 5]
```

⑦ arange() Function:-

- Used to create seq of numbers.
- Generate 'numpy.ndarray'

```
>>> np. arange (10)
```

O/p → [0 1 2 3 4 5 6 7 8 9]

$\boxed{\text{np.arange(start, stop, step, dtype)}}$

start → Default 0.

stop → provided.

step → Default 1.

dtype → Data type for array.
'f' - float
'i' - int

● Array creation using NumPy Functions :-

① **zeros** – create array with all elements 0, with given dimensions.

```
>>> a = np.zeros((2,3), dtype = int)
O/p →   [[0 0 0]
       2 [0 0 0]]
           3
```

```
np.zeros(shape = (), dtype = int,
                order = '')
```

shape → int / tuple of int for multi-dim array.

dtype → Data type or

order → {'C', 'F'}

'C' – row-major (C-style)

'F' – column-major (Fortan style)

② **ones** – Array with all elements 1, with dims.

```
>>> a = np.ones((2,3), dtype = int)
O/p →  [[1 1 1]
        [1 1 1]]
```

```
np.ones(shape, dtype, order)
```

shape → int/ tuple (multi-dim array)

③ **arange()** – create sequence of numbers.

```
>>> a = np.arange(10)
O/p → [0 1 2 3 4 5 6 7 8 9]
```

By default start from 0.

```
>>> b = np.arange(11, 21, dtype = int)
O/p → [11 12 13 14 15 16 17 18 19 20]
       start                      end-1
```

④ **linspace** – create 1-D array of linear space numbers / values, by default 50 linspace.

```
>>> p = np. linspace (3, 5)
```

O/p → [3.   3.04081 ....

....

.... 4.9591   5.]

50 nums array bet^n 3 & 5.

No. of samples to generate. Default 50.

```
linspace(start, stop, num, endpoint,
                retstep, dtype)
```

bool
If True, stop included, otherwise not. Default True.

If True, return tuple – (samples, step)

Spacing bet^n samples

Data type. Never int.

num

return step

```
>>> p = np.linspace(5, 2, 5, retstep = True)
```

start   stop

O/p → (array([5. , 4.25, 3.5, 2.75, 2.]), -0.75)

Array of 5 samples

step

end not included

num

```
>>> p = np.linspace( 2, 5, 5, endpoint = False,
                              retstep = True)
```

start   stop

O/p → (array([2. , 2.6, 3.2, 3.8, 4.4]), 0.6)

Array of 5 samples

Step

⑤ **eye** - Return 2D array with 1 at diagonal & 0 elsewhere.

```
np.eye(N, M, k, dtype, order)
```

N → rows

M → columns

k → Move diagonal.
+ve - move upward
-ve - move downward

dtype → Data type

order → {'C', 'F'}
'C' → row
'F' → column

```
>>> e = np.eye(4, 4, k=-1)
[[0.  0.  0.  0.]
 [1.  0.  0.  0.]
 [0.  1.  0.  0.]
 [0.  0.  1.  0.]]
```

4×4 & k=-1

-1 move 1 diagonal down.

```
[[1.  0.  0.  0.]
 [0.  1.  0.  0.]
 [0.  0.  1.  0.]
 [0.  0.  0.  1.]]
```

4×4

```
>>> e = np.eye(2, 3)
[[1.  0.  0.]
 [0.  1.  0.]]
```

2 → rows, 3 → cols

⑥ **identity** - Same as eye(), but take only 1 arg.
∴ row = column.

```
>>> d = np.identity(3)
[[1.  0.  0.]
 [0.  1.  0.]
 [0.  0.  1.]]
```

Generate square matrix with given int.

```
np.identity(n, dtype)
```

n → row = column

dtype → Data type.

⑦ **fromiter** — create ndarray from iterable obj:

↳ Return 1D ndarray object.

```
>>> lst = [0, 2, 4, 6, 8]
>>> it = iter(lst)
<list_iterator object at 0x000.....>

>>> x = np.fromiter(it, dtype = float)
O/p → [0.  2.  4.  6.]
      <class 'numpy.ndarray'>
```

⦿ Accessing Array Elements :-

① **Iterating array using nditer()** –

↳ Easy to iterate th' each scalar element in array, even for complex dimensions.

↳ i.e. iterate array like 1-D.

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

>>> for x in np.nditer(arr):
>>>     print(x)

O/p → 1
      2
      3
      4
      5
      6
```

Access ~~Create~~ array like 1-D.

② nditer() with step :-

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

>>> for x in np.nditer(arr[:, ::2]):
>>>     print(x)
```

O/p →

```
1
3
5
7
```

③ Iterate using ndenumerate() -

- ↳ enumerate - Mention sequence num one-by-one.
- ↳ Return index of elements while iterating.

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

>>> for idx, x in np.ndenumerate(arr):
>>>     print(idx, x)
```

O/p →

```
row col
(0, 0)  1
(0, 1)  2
(0, 2)  3
(1, 0)  4
(1, 1)  5
(1, 2)  6
```

Return tuple of index in multi-dim arr.

1) Array Indexing :-

① 1D array Indexing :- Indexing start from 0.

```
                0  1  2  3  4
>>> np.array ([1, 2, 3, 4, 5])

>>> arr [0]  ⟹ 1
```

② 2D Indexing - Provide comma-separated indices.

```
>>> a = np.array([[1,2,8], [4,5,6]])

>>> a[0, 1] ,  a[0][1]
```

$$\begin{array}{c|ccc} & 0 & 1 & 2 \\ \hline 0 & 1 & 2 & 3 \\ 1 & 4 & 5 & 6 \end{array}$$

O/P → 2

③ 3D Indexing - 3 indices.

```
>>> [[[1,2,3], [4,5,6]], [[7,8,9], [10,11,12]]]
```

$$\begin{array}{cc|ccc} 0 & 0 & 1 & 2 & 3 \\ & 1 & 4 & 5 & 6 \\ 1 & 0 & 7 & 8 & 9 \\ & 1 & 10 & 11 & 12 \\ \hline & & 0 & 1 & 2 \end{array}$$

```
>>> arr [0, 1, 2]

>>> arr [0][1][2]
```

O/P → 6

④ Negative Indexing - starts from the end.

```
>>> [[1, 2, 3, 4, 5], [6,7,8,9,10]]
```

$$\begin{array}{cc|ccccc} & & 0 & 1 & 2 & 3 & 4 \\ \hline 0 & -2 & 1 & 2 & 3 & 4 & 5 \\ 1 & -1 & 6 & 7 & 8 & 9 & 10 \\ \hline & & -5 & -4 & -3 & -2 & -1 \end{array}$$

```
>>> arr [-1, -1]
```

O/P → 10

2) Array Slicing:-

Access elements betn start & end indices.

```
arr [start : end]
```

```
arr [start : end : step]
```

→ end index is not included.

+ve →	0	1	2	3	4	5	6	7	8	
	1	2	3	4	5	6	7	8	9	
	-9	-8	-7	-6	-5	-4	-3	-2	-1	← -ve

```
>>> arr [1:5] → 5 not included

O/p →   [2  3  4  5]
         1  2  3  4
```

```
>>> arr [4:] → when end not given,
                Default len(arr).

O/p →  [5  6  7  8  9]
        4  5  6  7  8
        ─────────────→
        4 to end
```

- **Slice 2D Array:**

```
       0    1    2    3    4
0  [[1    2    3    4    5]

1   [6    7    8    9    10]]
```

```
                               1   2   3
>>> arr [1, 1:4]      →  1 [7   8   9]
                                0     1
>>> arr [0:2, 2]      →  2 [3     8]
                             1     2     3
>>> arr [0:2, 1:4]    →  0 [[2     3     4]
                          1  [7     8     9]]
```

⊙ Attributes of ndarray :-

```
a  =  [42]

b  =  [1,   2,   3,   4,   5]

c  =  [[1   2   3]
        [4   5   6]]

          0   1   2              0   1   2
d  =  [0[[1   2   3]          0[[7,  8   9]
       1 [4   5   6]],         1 [10  11  12]]]
            0                       1
```

① ndmin - Return dimensions of the array.

② shape - Return shape - (row, col) of array.

③ size - Return total num of elements in array. (row* col) OR multiplicaⁿ of shape.

	a	b	c	d
ndim	0	1	2	3
shape	( )	(5,)	(2, 3)	(2, 2, 3)
size	1	5	6 (2*3)	12 (2*2*3)

● Other NumPy Functions :-

① **insert ()** - Insert value at given place.

↳ changes not permanent, need other variable.

```
                          0  1  2  3  4
>>> a = np.array ([1, 2, 3, 4, 5])

>>> b = np.insert (a, 2, 100)

 O/p → [1  2  100  3  4  5]
```

② **append ()** - Insert value at array end.

```
>>> a = [1, 2, 3, 4, 5]

>>> b = np.append (a, 200)

 O/p → [1  2  3  4  5  200]
```

a = [1.2   5.8   9.4   5.8   12.5]

③ **ceil ()** - Yield to upper closest int for given float.

```
>>> a = np.ceil (a)

O/p → [2.  6.  10.  6.  13.]
```

floor()   ceil()

1 ← 1.2 → 2

5 ← 5.8 → 6

9 ← 9.4 → 10

5 ← 5.8 → 6

12 ← 12.5 → 13

④ **floor ()** – Yield closest lower int for float.

```
>>> a = np.floor (a)
```

O/p → [1.   5.   9.   5.   12.]

⑤ **around ()** – Yied to closest int.

i.e. floating point <.5 => ~~upper~~ lower

floating point >=.5 => upper

```
>>> np.around (a)
```

O/p → [1.   6.   9.   6.   13.]

1 ← 1.2

5.8 → 6

9 ← 9.4

5.8 → 6

12.5 → 13

⑥ **argmax ()** – Return index of max element in arr.

```
                    0    1   2  3  4  5
>>> arr = np.array ( [10, 15, 2, 1, 8, 16])
>>> np.argmax (arr)
```

O/p → 5 (Max element : 16)

⑦ **argmin ()** – Return index of min element in array.

```
>>> np.argmin (arr)
```

O/p → 3 (Min element : 1)

⑧ **size ()** – Return size of array.

```
>>> np.size (arr)
```

O/p → 6

⑨ **where ()** – Return indices of values, for which given condition is satisfied.

```
np. where(condition, True ( Replace this value),
                     False ( Replace this value))
```

```
               0      1     2     3       4      5
>>>  a =  [ 1.2,  5.8,  9.4,  5.6,  12.5,  4.8]

>>> np. where ( a > 5)
```

O/p → (array ([1, 2, 3, 4]), dtype = int64),)

```
>>> np. where ( a > 5,  10,  0)
```

Replace this val for condi'n == True

Replace condi'n == False

```
       1    2    3    4
[0    10   10   10   10   0] .
```

⊙ Random Number Generation :-

- **random Module** -
  - ↳ Built-in module for random num generation.
  - ↳ May not possess true randomness.

① **rand()** - Return 1D array of values betⁿ 0&1.

↳ Can provide (x, y) args to reshape.

```
>>> a = np.random.rand(10)
O/p →  [0.441 ..........  0.3046]
```

1D with size 10.

```
>>> b = np.random.rand(2, 3)
  2 { [[0.66     0.34     0.12
        [0.82     0.82     0.99]]
                  3
```

② **random()** - Return 1D array only, bcz 1 arg.

↳ Use reshape() afterwards.

```
>>> a = np.random.random(10)
   [0.01  0.63 ........  0.47] → 1D 10 size.

>>> a.reshape(2, 5)
   [[0.01   0.63 ........ ]
    [0.23 ............... ]]  } 2
                5
```

③ **ranf ()** – 1 arg, Random num bet^n 0 & 1.

```
>>> np.random.ranf (5)
```

O/p → [0.61   0.66   0.50   0.29   0.34]

④ **randint ()** – Random int numbers.

```
np.random.randint (low, high, size, dtype='i')
```

```
>>> np.random.randint (2, 10, 10, dtype = int)
```

O/p → $\underbrace{[7\ 7\ 6\ 7\ 2\ 3\ 8\ 9\ 1\ 5]}_{\text{size = 10}}$

Random nums bet^n 2(low) & 10(high)

```
>>> np.random.randint (2,10) => 6
```

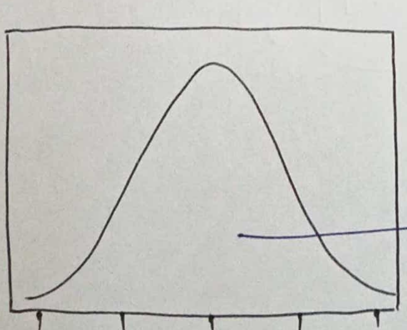
↳ size not given, generate only 1 num.

⑤ **randn ()** – Normally distributed nums around (0,0) i.e. Origin co-ordinates.

```
np.random.randn (row, col, ....)
```

```
>>> a = np.random.randn(2)
```

[-1.43   -0.55]



seaborn.kdeplot (a)

→ Normally distributed plot.