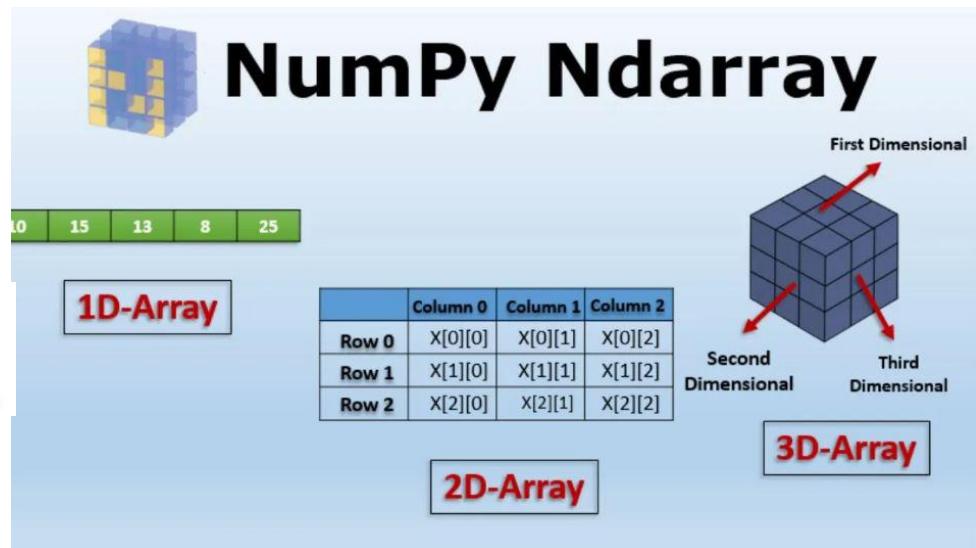


# NumPy: From Basics to Advanced

NumPy is a Python library that provides powerful and versatile array computations, mathematical functions, and other tools for various scientific domains. It is widely used in data science, machine learning, and scientific computing. NumPy offers comprehensive mathematical functions, random number generators, linear algebra routines, Fourier transforms, and more.



## Creating Numpy Arrays

```
In [1]: # np.array
import numpy as np

# create numpy 1D array
a = np.array([1, 2, 3, 4, 5])
print(a)

[1 2 3 4 5]

In [2]: # type of array
print(type(a))

<class 'numpy.ndarray'>

In [3]: #1D array of length 10 all values 0
np.zeros(10)

Out[3]: array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])

In [4]: #5x5 array with all values 1
np.ones((5,5))

Out[4]: 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., 1.]])
```

```
In [5]: #5x5 array of 0 with 1 on diagonal Identitymatrix
np.eye(5)

Out[5]: array([[1., 0., 0., 0., 0.],
               [0., 1., 0., 0., 0.],
               [0., 0., 1., 0., 0.],
               [0., 0., 0., 1., 0.],
               [0., 0., 0., 0., 1.]])
```

```
In [6]: #create 2D array
b = np.array([[1, 2, 3, 4, 5], [6, 7, 8, 9, 10]])
print(b)

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

```
In [7]: # create 3D array
c = np.array([[[1, 2, 3], [4,5,6]], [[7,8,9], [10,11,12]]])
print(c)

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

 [[ 7  8  9]
 [10 11 12]]]
```

```
In [8]: #here 1,15 is the range means give me 3 numbers having values between 1-15
np.random.randint(1,15,3)

Out[8]: array([11,  4,  7])
```

```
In [9]: #3x3 array with random ints between 0-9
np.random.randint(10,size=(3,3))

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

```
In [10]: # bool datatype
np.array([1, 2, 3, 4, 5], dtype=bool)

Out[10]: array([ True,  True,  True,  True,  True])
```

```
In [11]: #7x7 array of 0 with 1 on diagonal Identitymatrix
np.eye(7)

Out[11]: array([[1., 0., 0., 0., 0., 0., 0.],
 [0., 1., 0., 0., 0., 0., 0.],
 [0., 0., 1., 0., 0., 0., 0.],
 [0., 0., 0., 1., 0., 0., 0.],
 [0., 0., 0., 0., 1., 0., 0.],
 [0., 0., 0., 0., 0., 1., 0.],
 [0., 0., 0., 0., 0., 0., 1.]])
```

```
In [12]: #Array of values from 0 to less than 15 with step 3
np.arange(0,15,3)

Out[12]: array([ 0,  3,  6,  9, 12])
```

```
In [13]: from numpy import *
a = arange(12)
a = a.reshape(3,2,2)
print(a)

[[[ 0  1]
 [ 2  3]]

 [[ 4  5]
 [ 6  7]]

 [[ 8  9]
 [10 11]]]
```

```
In [14]: #Transposes arr (rows become columns and vice versa)
trans=np.array([(1,2,3,4),(5,6,7,8)])
trans.T
```

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

```
In [15]: trans.reshape(4,2)

Out[15]: array([[1, 2],
 [3, 4],
 [5, 6],
 [7, 8]])
```

```
In [16]: #Changes arr shape to 6x6 and fills new values with 0
trans.resize((6,6),refcheck=False)
trans
```

```
Out[16]: array([[1, 2, 3, 4, 5, 6],
 [7, 8, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0]])
```

```
In [17]: #Returns number of elements in arr  
arr=np.array([1,2,3,4,6,7,9])  
arr.size
```

Out[17]: 7

```
In [18]: #Appends values to end of arr  
arr2=np.append(arr,[0,1,2,4,5])  
arr2
```

```
Out[18]: array([1, 2, 3, 4, 6, 7, 9, 0, 1, 2, 4, 5])
```

```
In [19]: #Inserts values into arr before index 2  
np.insert(arr2,2,4)
```

```
Out[19]: array([1, 2, 4, 3, 4, 6, 7, 9, 0, 1, 2, 4, 5])
```

```
In [20]: #Deletes row on index 4 of arr  
np.delete(trans,4)
```

```
In [21]: #Deletes column on index 5 of array  
np.delete(trans,5, axis=1)
```

```
Out[21]: array([[1, 2, 3, 4, 5],  
                 [7, 8, 0, 0, 0],  
                 [0, 0, 0, 0, 0],  
                 [0, 0, 0, 0, 0],  
                 [0, 0, 0, 0, 0],  
                 [0, 0, 0, 0, 0]]))
```

In [22]: `abs(-1)`

Out[22]: 1

```
In [23]: add.accumulate(array([1.,2.,3.,4.]))
```

```
Out[23]: array([ 1.,  3.,  6., 10.])
```

```
In [24]: multiply.accumulate(array([1.,2.,3.,4.]))
```

```
Out[24]: array([ 1.,  2.,  6., 24.])
```

```
In [25]: add.accumulate(array([[1,2,3],[4,5,6]]), axis = 0)
```

```
Out[25]: array([[1, 2, 3],  
                 [5, 7, 9]])
```

```
In [26]: add.accumulate(array([[1,2,3],[4,5,6]]), axis = 1)
```

```
Out[26]: array([[ 1,  3,  6],  
                 [ 4,  9, 15]])
```

```
In [27]: add(array([-1.2, 1.2]), array([1,3]))
```

```
Out[27]: array([-0.2,  4.2])
```

```
In [28]: array([-1.2, 1.2]) + array([1,3])
```

Out[28]: array([-0.2, 4.2])

```
In [29]: a = array([True, False, True])
a.all()
```

Out[29]: False

```
In [30]: allclose(array([1e10,1e-7]), array([1.00001e10,1e-8]))
```

Out[30]: False

```
In [31]: # in radians  
angle(1+1j)
```

```
Out[31]: 0.7853981633974483
```

To: [redacted] # in

```
In [32]: angle(1+1j,deg=True)
Out[32]: 45.0
```

```
a = array([True, False, True])
a.any()
```

Out[33]: True

```
In [34]: from numpy import *
a = array([10,20,30,40])
append(a,50)
```

Out[34]: array([10, 20, 30, 40, 50])

```
In [35]: array([10, 20, 30, 40, 50])
append(a,[50,60])
```

Out[35]: array([10, 20, 30, 40, 50, 60])

```
In [36]: array([10, 20, 30, 40, 50])
append(a,[59,60])
```

Out[36]: array([10, 20, 30, 40, 59, 60])

```
In [37]: #View documentation for arr.tolist
np.info(arr.tolist)
```

a.tolist()

Return the array as an ``a.ndim``-levels deep nested list of Python scalars.

Return a copy of the array data as a (nested) Python list.  
Data items are converted to the nearest compatible builtin Python type, via  
the `~numpy.ndarray.item` function.

If ``a.ndim`` is 0, then since the depth of the nested list is 0, it will  
not be a list at all, but a simple Python scalar.

Parameters

-----

none

Returns

-----

y : object, or list of object, or list of list of object, or ...
The possibly nested list of array elements.

Notes

-----

The array may be recreated via ``a = np.array(a.tolist())``, although this  
may sometimes lose precision.

Examples

-----

For a 1D array, ``a.tolist()`` is almost the same as ``list(a)``,  
except that ``tolist`` changes numpy scalars to Python scalars:

```
>>> a = np.uint32([1, 2])
>>> a_list = list(a)
>>> a_list
[1, 2]
>>> type(a_list[0])
<class 'numpy.uint32'>
>>> a_tolist = a.tolist()
>>> a_tolist
[1, 2]
>>> type(a_tolist[0])
<class 'int'>
```

Additionally, for a 2D array, ``tolist`` applies recursively:

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

The base case for this recursion is a 0D array:

```
>>> a = np.array(1)
>>> list(a)
Traceback (most recent call last):
...
TypeError: iteration over a 0-d array
>>> a.tolist()
1
```

```
In [38]: from numpy import *
def myfunc(a): # function works on a 1d arrays, takes the average of the 1st an last element
    return (a[0]+a[-1])/2
```

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

```
Out[38]: array([4., 5., 6.])
```

```
In [39]: from numpy import *
a = arange(24).reshape(2,3,4) # a has 3 axes: 0,1 and 2
a
```

```
Out[39]: 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]]])
```

```
In [40]: # sum over all axes except axis=1, result has same shape as original
apply_over_axes(sum, a, [0,2])
```

```
Out[40]: array([[[ 60],
   [ 92],
   [124]]])
```

```
In [41]: from numpy import *
arange(5)
```

```
Out[41]: array([0, 1, 2, 3, 4])
```

```
In [42]: arange(5.0)
```

```
Out[42]: array([0., 1., 2., 3., 4.])
```

```
In [43]: from numpy import *
arccos(array([0, 1]))
```

```
Out[43]: array([1.57079633, 0.])
```

```
In [44]: arccosh(array([e, 10.0]))
```

```
Out[44]: array([1.65745445, 2.99322285])
```

```
In [45]: arcsin(array([0, 1]))
```

```
Out[45]: array([0., 1.57079633])
```

```
In [46]: arcsinh(array([e, 10.0]))
```

```
Out[46]: array([1.72538256, 2.99822295])
```

```
In [47]: arctan(array([0, 1]))
```

```
Out[47]: array([0., 0.78539816])
```

```
In [48]: arctan2(array([0, 1]), array([1, 0]))
```

```
Out[48]: array([0., 1.57079633])
```

```
In [49]: arctanh(array([0, -0.5]))
```

```
Out[49]: array([ 0., -0.54930614])
```

```
In [50]: from numpy import *
a = array([10,20,30])
maxindex = a.argmax()
a[maxindex]
```

```
Out[50]: 30
```

```
In [51]: a = array([[10,50,30],[60,20,40]])
maxindex = a.argmax()
maxindex
```

```
Out[51]: 3
```

```
In [52]: a.ravel()[maxindex]
```

```
Out[52]: 60
```

```
In [53]: # for each column: the row index of the maximum value
a.argmax(axis=0)
```

```
Out[53]: array([1, 0, 1], dtype=int64)
```

```
In [54]: # for each row: the column index of the maximum value  
a.argmax(axis=1)
```

```
Out[54]: array([1, 0], dtype=int64)
```

```
In [55]: # also exists, slower, default is axis=-1  
argmin(a)
```

```
Out[55]: 0
```

```
In [56]: from numpy import *  
a = array([2,0,8,4,1])  
a
```

```
Out[56]: array([2, 0, 8, 4, 1])
```

```
In [57]: # indices of sorted array using quicksort (default)  
ind = a.argsort()  
ind
```

```
Out[57]: array([1, 4, 0, 3, 2], dtype=int64)
```

```
In [58]: # same effect as a.sort()  
a[ind]
```

```
Out[58]: array([0, 1, 2, 4, 8])
```

```
In [59]: ind = a.argsort(kind='merge') # algorithm options are 'quicksort', 'mergesort' and 'heapsort'  
a = array([[8,4,1],[2,0,9]])  
ind = a.argsort(axis=0) # sorts on columns. NOT the same as a.sort(axis=1)  
ind
```

```
Out[59]: array([[1, 1, 0],  
               [0, 0, 1]], dtype=int64)
```

```
In [60]: from numpy import *  
abs(-1)
```

```
Out[60]: 1
```

```
In [61]: abs(array([-1.2, 1.2]))
```

```
Out[61]: array([1.2, 1.2])
```

```
In [62]: abs(1.2+1j)
```

```
Out[62]: 1.5620499351813308
```

```
In [63]: # like reduce() but also gives intermediate results  
from numpy import *  
add.accumulate(array([1.,2.,3.,4.]))
```

```
Out[63]: array([ 1.,  3.,  6., 10.])
```

```
In [64]: # works also with other operands  
multiply.accumulate(array([1.,2.,3.,4.]))
```

```
Out[64]: array([ 1.,  2.,  6., 24.])
```

```
In [65]: #Array of 5 evenly divided values from 0 to 150  
np.linspace(0,150,5)
```

```
Out[65]: array([ 0. ,  37.5,  75. , 112.5, 150. ])
```

```
In [66]: #Returns number of elements in arr  
arr=np.array([1,2,3,4,6,7,9])  
arr.size
```

```
Out[66]: 7
```

```
In [67]: #Returns dimensions of arr (rows,columns)  
arr.shape
```

```
Out[67]: (7,)
```

```
In [68]: #Convert arr to a Python list  
arr.tolist()
```

```
Out[68]: [1, 2, 3, 4, 6, 7, 9]
```

```
In [69]: #Copies arr to new memory
np.copy(arr)

Out[69]: array([1, 2, 3, 4, 6, 7, 9])
```

```
In [70]: #Sorts arr
arr.sort()
arr
```

```
Out[70]: array([1, 2, 3, 4, 6, 7, 9])
```

```
In [71]: #Sorts specific axis of arr
arr.sort(axis=0)
arr
```

```
Out[71]: array([1, 2, 3, 4, 6, 7, 9])
```

```
In [72]: #Appends values to end of arr
arr2=np.append(arr,[0,1,2,4,5])
arr2
```

```
Out[72]: array([1, 2, 3, 4, 6, 7, 9, 0, 1, 2, 4, 5])
```

```
In [73]: np.concatenate((arr,arr2),axis=0) #Adds arr2 as rows to the end of arr1
```

```
Out[73]: array([1, 2, 3, 4, 6, 7, 9, 1, 2, 3, 4, 6, 7, 9, 0, 1, 2, 4, 5])
```

```
In [74]: #Adds arr2 as columns to end of arr1
trans=np.array([(1,2,3,4),(5,6,7,8)])
np.concatenate((trans,trans),axis=1)
```

```
Out[74]: array([[1, 2, 3, 4, 1, 2, 3, 4],
 [5, 6, 7, 8, 5, 6, 7, 8]])
```

```
In [ ]:
```

```
In [ ]:
```

```
In [ ]:
```

```
Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js
```

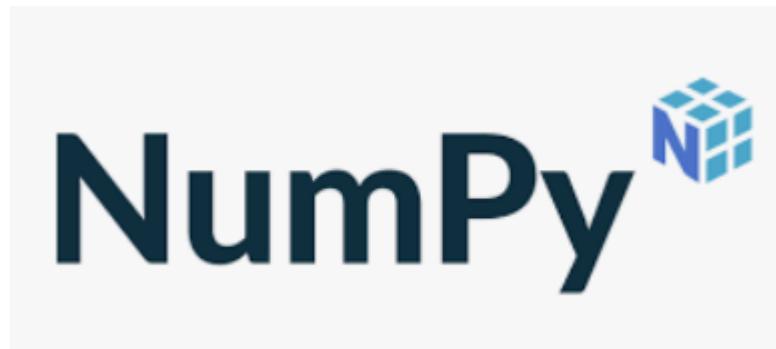
## Python Pandas From Basics to Advanced



- Prudhvi Vardhan Notes

## 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

## Creating Numpy array

```
In [1]: import numpy as np
```

```
In [2]: np.array([2,4,56,422,32,1]) # 1D array
```

```
Out[2]: array([ 2,  4, 56, 422, 32, 1])
```

```
In [3]: a = np.array([2,4,56,422,32,1]) #Vector
print(a)
```

```
[ 2  4 56 422 32 1]
```

```
In [4]: type(a)
```

```
Out[4]: numpy.ndarray
```

In [5]: # 2D Array ( Matrix )

```
new = np.array([[45,34,22,2],[24,55,3,22]])
print(new)
```

```
[[45 34 22  2]
 [24 55   3 22]]
```

In [6]: # 3 D ---- # Tensor

```
np.array ([[2,3,33,4,45],[23,45,56,66,2],[357,523,32,24,2],[32,32,44,33,234]])
```

Out[6]: array([[ 2, 3, 33, 4, 45],
 [ 23, 45, 56, 66, 2],
 [357, 523, 32, 24, 2],
 [ 32, 32, 44, 33, 234]])

## dtype

The desired data-type for the array. If not given, then the type will be determined as the minimum type required to hold the objects in the sequence.

In [7]: np.array([11,23,44] , dtype =float)

Out[7]: array([11., 23., 44.])

In [8]: np.array([11,23,44] , dtype =bool) # Here True becoz , python treats Non -zero as True

Out[8]: array([ True, True, True])

In [9]: np.array([11,23,44] , dtype =complex)

Out[9]: array([11.+0.j, 23.+0.j, 44.+0.j])

## 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.

## arange

arange can be called with a varying number of positional arguments

```
In [10]: np.arange(1,25) # 1-included , 25 - Last one got excluded
```

```
Out[10]: array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16, 17,
   18, 19, 20, 21, 22, 23, 24])
```

```
In [11]: np.arange(1,25,2) #strides ---> Alternate numbers
```

```
Out[11]: array([ 1,  3,  5,  7,  9, 11, 13, 15, 17, 19, 21, 23])
```

## reshape

Both of number products should be equal to umber of Items present inside the array.

```
In [12]: np.arange(1,11).reshape(5,2) # converted 5 rows and 2 columns
```

```
Out[12]: array([[ 1,  2],
   [ 3,  4],
   [ 5,  6],
   [ 7,  8],
   [ 9, 10]])
```

```
In [13]: np.arange(1,11).reshape(2,5) # converted 2 rows and 5 columns
```

```
Out[13]: array([[ 1,  2,  3,  4,  5],
   [ 6,  7,  8,  9, 10]])
```

```
In [14]: np.arange(1,13).reshape(3,4) # converted 3 rows and 4 columns
```

```
Out[14]: array([[ 1,  2,  3,  4],
   [ 5,  6,  7,  8],
   [ 9, 10, 11, 12]])
```

## ones & Zeros

you can initialize the values and create values . ex: in deep learning weight shape

```
In [15]: # np.ones and np.zeros
```

```
np.ones((3,4)) # we have to mention inside tuple
```

```
Out[15]: array([[1., 1., 1., 1.],
   [1., 1., 1., 1.],
   [1., 1., 1., 1.]])
```

In [16]: `np.zeros((3,4))`

Out[16]: `array([[0., 0., 0., 0.],  
[0., 0., 0., 0.],  
[0., 0., 0., 0.]])`

In [17]: `# Another Type ---> random()`

`np.random.random((4,3))`

Out[17]: `array([[0.36101914, 0.04882035, 0.23266312],  
[0.74023073, 0.01298753, 0.03403761],  
[0.80722213, 0.55568178, 0.94063313],  
[0.45455407, 0.06724469, 0.75013537]])`

## linspace

It is also called as Linearly space , Linearly separable,in a given range at equal distance it creates points.

In [18]: `np.linspace(-10,10,10) # here: Lower range,upper range ,number of items to gen`

Out[18]: `array([-10. , -7.77777778, -5.55555556, -3.33333333, -1.11111111, 1.11111111, 3.33333333, 5.55555556, 7.77777778, 10. ])`

In [19]: `np.linspace(-2,12,6)`

Out[19]: `array([-2. , 0.8, 3.6, 6.4, 9.2, 12. ])`

## identity

identity matrix is that diagonal items will be ones and everything will be zeros

In [20]: `# creating the identity matrix`

`np.identity(3)`

Out[20]: `array([[1., 0., 0.],  
[0., 1., 0.],  
[0., 0., 1.]])`

```
In [21]: np.identity(6)
```

```
Out[21]: array([[1., 0., 0., 0., 0., 0.],
   [0., 1., 0., 0., 0., 0.],
   [0., 0., 1., 0., 0., 0.],
   [0., 0., 0., 1., 0., 0.],
   [0., 0., 0., 0., 1., 0.],
   [0., 0., 0., 0., 0., 1.]])
```

## Array Attributes

```
In [22]: a1 = np.arange(10) # 1D
a1
```

```
Out[22]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
In [23]: a2 = np.arange(12, dtype = float).reshape(3,4) # Matrix
a2
```

```
Out[23]: array([[ 0.,  1.,  2.,  3.],
   [ 4.,  5.,  6.,  7.],
   [ 8.,  9., 10., 11.]])
```

```
In [24]: a3 = np.arange(8).reshape(2,2,2) # 3D --> Tensor
a3
```

```
Out[24]: array([[[0, 1],
   [2, 3]],
  [[[4, 5],
   [6, 7]]]])
```

### **ndim**

To findout given arrays number of dimensions

```
In [25]: a1.ndim
```

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

```
In [26]: a2.ndim
```

```
Out[26]: 2
```

```
In [27]: a3.ndim
```

```
Out[27]: 3
```

## shape

gives each item consist of no.of rows and np.of column

```
In [28]: a1.shape # 1D array has 10 Items
```

```
Out[28]: (10,)
```

```
In [29]: a2.shape # 3 rows and 4 columns
```

```
Out[29]: (3, 4)
```

```
In [30]: a3.shape # first ,2 says it consists of 2D arrays .2,2 gives no.of rows and c
```

```
Out[30]: (2, 2, 2)
```

## size

gives number of items

```
In [31]: a3
```

```
Out[31]: array([[[0, 1],
                  [2, 3]],
                 [[4, 5],
                  [6, 7]])
```

```
In [32]: a3.size # it has 8 items . Like shape :2,2,2 = 8
```

```
Out[32]: 8
```

```
In [33]: a2
```

```
Out[33]: array([[ 0.,  1.,  2.,  3.],
                 [ 4.,  5.,  6.,  7.],
                 [ 8.,  9., 10., 11.]])
```

```
In [34]: a2.size
```

```
Out[34]: 12
```

## item size

Memory occupied by the item

In [35]: a1

Out[35]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

In [36]: a1.itemsize # bytes

Out[36]: 4

In [37]: a2.itemsize # integer 64 gives = 8 bytes

Out[37]: 8

In [38]: a3.itemsize # integer 32 gives = 4 bytes

Out[38]: 4

## dtype

gives data type of the item

In [39]: print(a1.dtype)  
print(a2.dtype)  
print(a3.dtype)

int32  
float64  
int32

## Changing Data Type

In [40]: #astype

```
x = np.array([33, 22, 2.5])
x
```

Out[40]: array([33., 22., 2.5])

In [41]: x.astype(int)

Out[41]: array([33, 22, 2])

## Array operations

In [42]: z1 = np.arange(12).reshape(3,4)
z2 = np.arange(12,24).reshape(3,4)

In [43]: z1

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

In [44]: z2

```
Out[44]: array([[12, 13, 14, 15],
   [16, 17, 18, 19],
   [20, 21, 22, 23]])
```

## scalar operations

Scalar operations on Numpy arrays include performing addition or subtraction, or multiplication on each element of a Numpy array.

In [45]: # arithmetic  
z1 + 2

```
Out[45]: array([[ 2,  3,  4,  5],
   [ 6,  7,  8,  9],
   [10, 11, 12, 13]])
```

In [46]: # Subtraction  
z1 - 2

```
Out[46]: array([[-2, -1,  0,  1],
   [ 2,  3,  4,  5],
   [ 6,  7,  8,  9]])
```

In [47]: # Multiplication  
z1 \* 2

```
Out[47]: array([[ 0,  2,  4,  6],
   [ 8, 10, 12, 14],
   [16, 18, 20, 22]])
```

In [48]: # power  
z1 \*\* 2

```
Out[48]: array([[  0,   1,   4,   9],
   [ 16,  25,  36,  49],
   [ 64,  81, 100, 121]], dtype=int32)
```

In [49]: ## Modulo  
z1 % 2

```
Out[49]: array([[0, 1, 0, 1],
   [0, 1, 0, 1],
   [0, 1, 0, 1]], dtype=int32)
```

## relational Operators

The relational operators are also known as **comparison operators**, their main function is to return either a true or false based on the value of operands.

In [50]: z2

Out[50]: array([[12, 13, 14, 15],  
[16, 17, 18, 19],  
[20, 21, 22, 23]])

In [51]: z2 > 2 # if 2 is greater than evrything gives True

Out[51]: array([[ True, True, True, True],  
[ True, True, True, True],  
[ True, True, True, True]])

In [52]: z2 > 20

Out[52]: array([[False, False, False, False],  
[False, False, False, False],  
[False, True, True, True]])

## Vector Operation

We can apply on both numpy array

In [53]: z1

Out[53]: array([[ 0, 1, 2, 3],  
[ 4, 5, 6, 7],  
[ 8, 9, 10, 11]])

In [54]: z2

Out[54]: array([[12, 13, 14, 15],  
[16, 17, 18, 19],  
[20, 21, 22, 23]])

In [55]: # Arthematic

z1 + z2 # both numpy array Shape is same , we can add item wise

Out[55]: array([[12, 14, 16, 18],  
[20, 22, 24, 26],  
[28, 30, 32, 34]])

In [56]: `z1 * z2`

Out[56]: `array([[ 0, 13, 28, 45],  
 [ 64, 85, 108, 133],  
 [160, 189, 220, 253]])`

In [57]: `z1 - z2`

Out[57]: `array([[-12, -12, -12, -12],  
 [-12, -12, -12, -12],  
 [-12, -12, -12, -12]])`

In [58]: `z1 / z2`

Out[58]: `array([[0. , 0.07692308, 0.14285714, 0.2 ],  
 [0.25 , 0.29411765, 0.33333333, 0.36842105],  
 [0.4 , 0.42857143, 0.45454545, 0.47826087]])`

## Array Functions

In [59]: `k1 = np.random.random((3,3))  
k1 = np.round(k1*100)  
k1`

Out[59]: `array([[44., 98., 47.],  
 [56., 49., 30.],  
 [60., 54., 24.]])`

In [60]: `# Max  
np.max(k1)`

Out[60]: `98.0`

In [61]: `# min  
np.min(k1)`

Out[61]: `24.0`

In [62]: `# sum  
np.sum(k1)`

Out[62]: `462.0`

In [63]: `# prod ----> Multiplication  
np.prod(k1)`

Out[63]: `1297293445324800.0`

## In Numpy

0 = column , 1 = row

```
In [64]: # if we want maximum of every row  
np.max(k1, axis = 1)
```

```
Out[64]: array([98., 56., 60.])
```

```
In [65]: # maximum of every column  
np.max(k1, axis = 0)
```

```
Out[65]: array([60., 98., 47.])
```

```
In [66]: # product of every column  
np.prod(k1, axis = 0)
```

```
Out[66]: array([147840., 259308., 33840.])
```

## Statistics related functions

```
In [67]: # mean  
k1
```

```
Out[67]: array([[44., 98., 47.],  
                 [56., 49., 30.],  
                 [60., 54., 24.]])
```

```
In [68]: np.mean(k1)
```

```
Out[68]: 51.33333333333336
```

```
In [69]: # mean of every column  
k1.mean(axis=0)
```

```
Out[69]: array([53.33333333, 67. , 33.66666667])
```

```
In [70]: # median  
np.median(k1)
```

```
Out[70]: 49.0
```

```
In [71]: np.median(k1, axis = 1)
```

```
Out[71]: array([47., 49., 54.])
```

In [72]: *# Standard deviation*

```
np.std(k1)
```

Out[72]: 19.89416441516903

In [73]: np.std(k1, axis =0)

Out[73]: array([ 6.79869268, 22.0151463 , 9.7410928 ])

In [74]: *# variance*

```
np.var(k1)
```

Out[74]: 395.77777777777777

## Trigonometry Functions

In [75]: np.sin(k1) *# sin*

Out[75]: array([[ 0.01770193, -0.57338187, 0.12357312],  
[-0.521551 , -0.95375265, -0.98803162],  
[-0.30481062, -0.55878905, -0.90557836]])

In [76]: np.cos(k1)

Out[76]: array([[ 0.99984331, -0.81928825, -0.99233547],  
[ 0.85322011, 0.30059254, 0.15425145],  
[-0.95241298, -0.82930983, 0.42417901]])

In [77]: np.tan(k1)

Out[77]: array([[ 0.0177047 , 0.69985365, -0.12452757],  
[-0.61127369, -3.17290855, -6.4053312 ],  
[ 0.32004039, 0.6738001 , -2.1348967 ]])

## dot product

The numpy module of Python provides a function to perform the dot product of two arrays.

In [78]: s2 = np.arange(12).reshape(3,4)  
s3 = np.arange(12,24).reshape(4,3)

In [79]: s2

Out[79]: array([[ 0, 1, 2, 3],  
[ 4, 5, 6, 7],  
[ 8, 9, 10, 11]])

In [80]: s3

```
Out[80]: array([[12, 13, 14],
 [15, 16, 17],
 [18, 19, 20],
 [21, 22, 23]])
```

In [81]: np.dot(s2,s3) # dot product of s2 , s3

```
Out[81]: array([[114, 120, 126],
 [378, 400, 422],
 [642, 680, 718]])
```

## Log and Exponents

In [82]: np.exp(s2)

```
Out[82]: array([[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]])
```

## round / floor /ceil

### 1. round

The numpy.round() function rounds the elements of an array to the nearest integer or to the specified number of decimals.

In [87]: # Round to the nearest integer  
arr = np.array([1.2, 2.7, 3.5, 4.9])  
rounded\_arr = np.round(arr)  
print(rounded\_arr)

[1. 3. 4. 5.]

In [88]: # Round to two decimals  
arr = np.array([1.234, 2.567, 3.891])  
rounded\_arr = np.round(arr, decimals=2)  
print(rounded\_arr)

[1.23 2.57 3.89]

In [84]: #randomly  
np.round(np.random.random((2,3))\*100)

```
Out[84]: array([[ 8., 36., 43.],
 [13., 90., 63.]])
```

### 2. floor

The numpy.floor() function returns the largest integer less than or equal to each element of an array.

```
In [89]: # Floor operation
arr = np.array([1.2, 2.7, 3.5, 4.9])
floored_arr = np.floor(arr)
print(floored_arr)
```

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

```
In [85]: np.floor(np.random.random((2,3))*100) # gives the smallest integer ex :6.8 =
```

```
Out[85]: array([[58., 56., 89.],
 [10., 83., 34.]])
```

### 3. Ceil

The numpy.ceil() function returns the smallest integer greater than or equal to each element of an array.

```
In [90]: arr = np.array([1.2, 2.7, 3.5, 4.9])
ceiled_arr = np.ceil(arr)
print(ceiled_arr)
```

```
[2. 3. 4. 5.]
```

```
In [86]: np.ceil(np.random.random((2,3))*100) # gives highest integer ex : 7.8 = 8
```

```
Out[86]: array([[94., 5., 46.],
 [84., 71., 41.]])
```

## Indexing and slicing

```
In [91]: p1 = np.arange(10)
p2 = np.arange(12).reshape(3,4)
p3 = np.arange(8).reshape(2,2,2)
```

```
In [92]: p1
```

```
Out[92]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
In [93]: p2
```

```
Out[93]: array([[ 0,  1,  2,  3],
 [ 4,  5,  6,  7],
 [ 8,  9, 10, 11]])
```

```
In [94]: p3
```

```
Out[94]: array([[ [0, 1],  
                   [2, 3],  
  
                   [[4, 5],  
                    [6, 7]]]])
```

### Indexing on 1D array

```
In [95]: p1
```

```
Out[95]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
In [96]: # fetching last item  
  
p1[-1]
```

```
Out[96]: 9
```

```
In [97]: # fetching first item  
  
p1[0]
```

```
Out[97]: 0
```

### indexing on 2D array

```
In [98]: p2
```

```
Out[98]: array([[ 0,  1,  2,  3],  
                 [ 4,  5,  6,  7],  
                 [ 8,  9, 10, 11]])
```

```
In [100]: # fetching desired element : 6  
  
p2[1,2] # here 1 = row(second) , 2= column(third) , becoz it starts from zero
```

```
Out[100]: 6
```

```
In [101]: # fetching desired element : 11  
  
p2[2,3] # row =2 , column =3
```

```
Out[101]: 11
```

```
In [102]: # fetching desired element : 4
          p2[1,0] # row =1 , column =0
```

Out[102]: 4

### indexing on 3D ( Tensors)

```
In [103]: p3
```

```
Out[103]: array([[[0, 1],
                   [2, 3],
                   [4, 5],
                   [6, 7]]])
```

```
In [106]: # fetching desired element : 5
          p3[1,0,1]
```

Out[106]: 5

EXPLANATION :Here 3D is consists of 2 ,2D array , so Firstly we take 1 because our desired is 5 is in second matrix which is 1 .and 1 row so 0 and second column so 1

```
In [109]: # fetching desired element : 2
          p3[0,1,0]
```

Out[109]: 2

EXPLANATION :Here firstly we take 0 because our desired is 2, is in first matrix which is 0 . and 2 row so 1 and first column so 0

```
In [110]: # fetching desired element : 0
          p3[0,0,0]
```

Out[110]: 0

Here first we take 0 because our desired is 0, is in first matrix which is 0 . and 1 row so 0 and first column so 0

```
In [113]: # fetching desired element : 6
          p3[1,1,0]
```

Out[113]: 6

**EXPLANATION :** Here first we take because our desired is 6, is in second matrix which is 1 . and second row so 1 and first column so 0

## Slicing

Fetching Multiple items

### Slicing on 1D

```
In [114]: p1
```

```
Out[114]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
In [116]: # fetching desired elements are : 2,3,4
```

```
p1[2:5]
```

```
Out[116]: array([2, 3, 4])
```

**EXPLANATION :** Here First we take , whatever we need first item ,2 and up last(4) + 1 which 5 .because last element is not included

```
In [117]: # Alternate (same as python)
```

```
p1[2:5:2]
```

```
Out[117]: array([2, 4])
```

### Slicing on 2D

```
In [121]: p2
```

```
Out[121]: array([[ 0,  1,  2,  3],
                  [ 4,  5,  6,  7],
                  [ 8,  9, 10, 11]])
```

```
In [122]: # fetching total First row
```

```
p2[0, :]
```

```
Out[122]: array([0, 1, 2, 3])
```

**EXPLANATION :** Here 0 represents first row and (:) represents Total column

In [124]: # fetching total third column

```
p2[:,2]
```

Out[124]: array([ 2, 6, 10])

EXPLANATION :Here we want all rows so () , and we want 3rd column so 2

In [164]: # fetch 5,6 and 9,10

```
p2
```

Out[164]: array([[ 0, 1, 2, 3],  
 [ 4, 5, 6, 7],  
 [ 8, 9, 10, 11]])

In [165]: p2[1:3] # for rows

Out[165]: array([[ 4, 5, 6, 7],  
 [ 8, 9, 10, 11]])

In [127]: p2[1:3 ,1:3] # For columns

Out[127]: array([[ 5, 6],  
 [ 9, 10]])

EXPLANATION :Here first [1:3] we slice 2 second row is to third row is not existed which is 2 and Secondly , we take [1:3] which is same as first:we slice 2 second row is to third row is not included which is 3

In [129]: # fetch 0,3 and 8,11

```
p2
```

Out[129]: array([[ 0, 1, 2, 3],  
 [ 4, 5, 6, 7],  
 [ 8, 9, 10, 11]])

In [130]: p2[::-2, ::3]

Out[130]: array([[ 0, 3],  
 [ 8, 11]])

EXPLANATION : Here we take () because we want all rows , second(:2) for alternate value, and () for all columns and (:3) jump for two steps

In [163]: `# fetch 1,3 and 9,11`

p2

Out[163]: `array([[ 0, 1, 2, 3],  
 [ 4, 5, 6, 7],  
 [ 8, 9, 10, 11]])`

In [162]: `p2[::-2] # For rows`

Out[162]: `array([[ 0, 1, 2, 3],  
 [ 8, 9, 10, 11]])`

In [ ]: `p2[::-2 ,1::2] # columns`

EXPLANATION : Here we take `(:)` because we want all rows , second(`:2`) for alternate value, and `(1)` for we want from second column and `(:2)` jump for two steps and ignore middle one

In [160]: `# fetch only 4 ,7`

p2

Out[160]: `array([[ 0, 1, 2, 3],  
 [ 4, 5, 6, 7],  
 [ 8, 9, 10, 11]])`

In [161]: `p2[1] # first rows`

Out[161]: `array([4, 5, 6, 7])`

In [150]: `p2[1,:,:3] # second columns`

Out[150]: `array([4, 7])`

EXPLANATION : Here we take `(1)` because we want second row , second(`:`) for total column, `(:3)` jump for two steps and ignore middle ones

In [157]: `# fetch 1,2,3 and 5,6,7`  
p2

Out[157]: `array([[ 0, 1, 2, 3],  
 [ 4, 5, 6, 7],  
 [ 8, 9, 10, 11]])`

In [159]: `p2[0:2] # first fetched rows`

Out[159]: `array([[0, 1, 2, 3],  
 [4, 5, 6, 7]])`

```
In [156]: p2[0:2 ,1: ] # for column
```

```
Out[156]: array([[1, 2, 3],
 [5, 6, 7]])
```

```
In [166]: # fetch 1,3 and 5,7
p2
```

```
Out[166]: array([[ 0,  1,  2,  3],
 [ 4,  5,  6,  7],
 [ 8,  9, 10, 11]])
```

```
In [167]: p2[0:2] # for rows
```

```
Out[167]: array([[0, 1, 2, 3],
 [4, 5, 6, 7]])
```

```
In [170]: p2[0:2 ,1::2]
```

```
Out[170]: array([[1, 3],
 [5, 7]])
```

**EXPLANATION :** 0:2 selects the rows from index 0 (inclusive) to index 2 (exclusive), which means it will select the first and second rows of the array. , is used to separate row and column selections. 1::2 selects the columns starting from index 1 and selects every second column. So it will select the second and fourth columns of the array.

## Slicing in 3D

```
In [172]: p3 = np.arange(27).reshape(3,3,3)
p3
```

```
Out[172]: 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 [173]: # fetch second matrix
p3[1]
```

```
Out[173]: array([[ 9, 10, 11],
 [12, 13, 14],
 [15, 16, 17]])
```

In [179]: # fetch first and last

```
p3[::-2]
```

Out[179]: array([[[ 0, 1, 2],  
 [ 3, 4, 5],  
 [ 6, 7, 8]],  
  
 [[18, 19, 20],  
 [21, 22, 23],  
 [24, 25, 26]]])

EXPLANATION : Along the first axis, (:) selects every second element. This means it will select the subarrays at indices 0 and 2

In [180]: # Fetch 1 2d array's 2 row ---> 3,4,5

```
p3
```

Out[180]: 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 [185]: p3[0] # first numpy array

Out[185]: array([[0, 1, 2],  
 [3, 4, 5],  
 [6, 7, 8]])

In [186]: p3[0,1,:]

Out[186]: array([3, 4, 5])

EXPLANATION : 0 represents first matrix , 1 represents second row , (:) means total

In [187]: # Fetch 2 numpy array ,middle column ---> 10,13,16

p3

Out[187]: 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 [189]: p3[1] # middle Array

Out[189]: array([[ 9, 10, 11],  
               [12, 13, 14],  
               [15, 16, 17]])

In [191]: p3[1,:,:1]

Out[191]: array([10, 13, 16])

EXPLANATION : 1 respresents middle column , (:) all columns , 1 represents middle column

In [192]: # Fetch 3 array--->22,23,25,26

p3

Out[192]: 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 [194]: p3[2] # Last row

Out[194]: array([[18, 19, 20],  
               [21, 22, 23],  
               [24, 25, 26]])

In [195]: `p3[2, 1:] # last two rows`

Out[195]: `array([[21, 22, 23],  
[24, 25, 26]])`

In [196]: `p3[2, 1: ,1:] # last two columns`

Out[196]: `array([[22, 23],  
[25, 26]])`

EXPLANATION : Here we go through 3 stages , where 2 for last array , and (1:) from second row to total rows , and (1:) is for second column to total columns

In [197]: `# Fetch o, 2, 18 , 20  
p3`

Out[197]: `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 [201]: `p3[0::2] # for arrays`

Out[201]: `array([[[ 0, 1, 2],  
[ 3, 4, 5],  
[ 6, 7, 8]],  
[[18, 19, 20],  
[21, 22, 23],  
[24, 25, 26]]])`

In [206]: `p3[0::2 , 0] # for rows`

Out[206]: `array([[ 0, 1, 2],  
[18, 19, 20]])`

In [207]: `p3[0::2 , 0 , ::2] # for columns`

Out[207]: `array([[ 0, 2],  
[18, 20]])`

EXPLANATION : Here we take (0::2) first adn last column , so we did jump using this, and we took (0) for first row , and we (::2) ignored middle column

## Iterating

In [208]: p1

Out[208]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

In [211]: # Looping on 1D array

```
for i in p1:  
    print(i)
```

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

In [209]: p2

Out[209]: array([[ 0, 1, 2, 3],  
 [ 4, 5, 6, 7],  
 [ 8, 9, 10, 11]])

In [212]: ## Looping on 2D array

```
for i in p2:  
    print(i) # prints rows
```

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

In [210]: p3

Out[210]: 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 [213]: for i in p3:  
    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]]
```

print all items in 3D using **nditer** ----> first convert in to 1D and applying Loop

```
In [215]: for i in np.nditer(p3):  
    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
```

## Reshaping

**Transpose** ---> Converts rows in to columns ad columns into rows

In [217]: p2

```
Out[217]: array([[ 0,  1,  2,  3],
   [ 4,  5,  6,  7],
   [ 8,  9, 10, 11]])
```

In [219]: np.transpose(p2)

```
Out[219]: array([[ 0,  4,  8],
   [ 1,  5,  9],
   [ 2,  6, 10],
   [ 3,  7, 11]])
```

In [222]: *# Another method*

```
p2.T
```

```
Out[222]: array([[ 0,  4,  8],
   [ 1,  5,  9],
   [ 2,  6, 10],
   [ 3,  7, 11]])
```

In [221]: p3

```
Out[221]: 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 [223]: p3.T

```
Out[223]: array([[[[ 0,  9, 18],
   [ 3, 12, 21],
   [ 6, 15, 24]],

   [[[ 1, 10, 19],
   [ 4, 13, 22],
   [ 7, 16, 25]],

   [[[ 2, 11, 20],
   [ 5, 14, 23],
   [ 8, 17, 26]]]])
```

## Ravel

Converting any dimensions to 1D

In [225]: p2

```
Out[225]: array([[ 0,  1,  2,  3],
                  [ 4,  5,  6,  7],
                  [ 8,  9, 10, 11]])
```

In [224]: p2.ravel()

```
Out[224]: array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])
```

In [226]: p3

```
Out[226]: 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 [227]: p3.ravel()

```
Out[227]: 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])
```

## Stacking

Stacking is the concept of joining arrays in NumPy. Arrays having the same dimensions can be stacked

In [230]: # Horizontal stacking

```
w1 = np.arange(12).reshape(3,4)
w2 = np.arange(12,24).reshape(3,4)
```

In [231]: w1

```
Out[231]: array([[ 0,  1,  2,  3],
                  [ 4,  5,  6,  7],
                  [ 8,  9, 10, 11]])
```

In [232]: w2

```
Out[232]: array([[12, 13, 14, 15],
                  [16, 17, 18, 19],
                  [20, 21, 22, 23]])
```

using **hstack** for Horizontal stacking

```
In [236]: np.hstack((w1,w2))
```

```
Out[236]: 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 [237]: # Vertical stacking
w1
```

```
Out[237]: array([[ 0,  1,  2,  3],
   [ 4,  5,  6,  7],
   [ 8,  9, 10, 11]])
```

```
In [238]: w2
```

```
Out[238]: array([[12, 13, 14, 15],
   [16, 17, 18, 19],
   [20, 21, 22, 23]])
```

using **vstack** for vertical stacking

```
In [239]: np.vstack((w1,w2))
```

```
Out[239]: 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

its opposite of Stacking .

```
In [240]: # Horizontal splitting
```

```
w1
```

```
Out[240]: array([[ 0,  1,  2,  3],
   [ 4,  5,  6,  7],
   [ 8,  9, 10, 11]])
```

```
In [241]: np.hsplit(w1,2) # splitting by 2
```

```
Out[241]: [array([[0, 1],  
                   [4, 5],  
                   [8, 9]]),  
           array([[ 2,  3],  
                  [ 6,  7],  
                  [10, 11]])]
```

```
In [242]: np.hsplit(w1,4) # splitting by 4
```

```
Out[242]: [array([[0],  
                   [4],  
                   [8]]),  
           array([[1],  
                  [5],  
                  [9]]),  
           array([[ 2],  
                  [ 6],  
                  [10]]),  
           array([[ 3],  
                  [ 7],  
                  [11]])]
```

```
In [244]: # Vertical splitting
```

```
w2
```

```
Out[244]: array([[12, 13, 14, 15],  
                  [16, 17, 18, 19],  
                  [20, 21, 22, 23]])
```

```
In [246]: np.vsplit(w2,3) # splitting into 3 rows
```

```
Out[246]: [array([[12, 13, 14, 15]]),  
           array([[16, 17, 18, 19]]),  
           array([[20, 21, 22, 23]])]
```

```
In [ ]:
```

## 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.

## Speed of List Vs Numpy

### List

In [1]: # Element-wise addition

```
a = [ i for i in range(10000000)]
b = [i for i in range(10000000,20000000)]

c = []

import time

start = time.time()
for i in range(len(a)):
    c.append(a[i] + b[i])

print(time.time()-start)
```

2.0619215965270996

## Numpy

In [2]: import numpy as np

```
a = np.arange(10000000)
b = np.arange(10000000,20000000)

start = time.time()
c = a+b
print(time.time()-start)
```

0.1120920181274414

In [3]: 2.7065064907073975 / 0.02248692512512207

Out[3]: 120.35911871666826

so ,**Numpy** is Faster than Normal Python programming ,we can see in above Example.  
because Numpy uses C type array

## Memory Used for List Vs Numpy

### List

In [4]: P = [i for i in range(10000000)]

```
import sys
sys.getsizeof(P)
```

Out[4]: 89095160

### Numpy

```
In [5]: R = np.arange(10000000)
```

```
sys.getsizeof(R)
```

```
Out[5]: 40000104
```

```
In [6]: # we can decrease more in numpy
```

```
R = np.arange(10000000, dtype =np.int16)
```

```
sys.getsizeof(R)
```

```
Out[6]: 20000104
```

## Advance Indexing and Slicing

```
In [7]: # Normal Indexing and slicing
```

```
w = np.arange(12).reshape(4,3)
```

```
w
```

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

```
In [8]: # Fetching 5 from array
```

```
w[1,2]
```

```
Out[8]: 5
```

```
In [9]: # Fetching 4,5,7,8
```

```
w[1:3]
```

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

```
In [10]: w[1:3 , 1:3]
```

```
Out[10]: array([[4, 5],
   [7, 8]])
```

## Fancy Indexing

Fancy indexing allows you to select or modify specific elements based on complex conditions or combinations of indices. It provides a powerful way to manipulate array data in NumPy.

```
In [11]: w
```

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

```
In [12]: # Fetch 1,3,4 row
```

```
w[[0,2,3]]
```

```
Out[12]: array([[ 0,  1,  2],  
                 [ 6,  7,  8],  
                 [ 9, 10, 11]])
```

```
In [13]: # New array
```

```
z = np.arange(24).reshape(6,4)  
z
```

```
Out[13]: 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]])
```

```
In [14]: # Fetch 1, 3, ,4, 6 rows
```

```
z[[0,2,3,5]]
```

```
Out[14]: array([[ 0,  1,  2,  3],  
                 [ 8,  9, 10, 11],  
                 [12, 13, 14, 15],  
                 [20, 21, 22, 23]])
```

```
In [15]: # Fetch 1,3,4 columns
```

```
z[:,[0,2,3]]
```

```
Out[15]: array([[ 0,  2,  3],  
                 [ 4,  6,  7],  
                 [ 8, 10, 11],  
                 [12, 14, 15],  
                 [16, 18, 19],  
                 [20, 22, 23]])
```

## Boolean indexing

It allows you to select elements from an array based on a **Boolean condition**. This allows you to extract only the elements of an array that meet a certain condition, making it easy to perform operations on specific subsets of data.

```
In [16]: G = np.random.randint(1,100,24).reshape(6,4)
```

```
In [17]: G
```

```
Out[17]: array([[64, 51, 75, 50],
 [ 8, 86,  6, 53],
 [60, 50, 49, 95],
 [75, 79, 98, 34],
 [45, 35, 87, 58],
 [56, 26, 93, 17]])
```

```
In [18]: # find all numbers greater than 50
```

```
G > 50
```

```
Out[18]: array([[ True,  True,  True, False],
 [False,  True, False,  True],
 [ True, False, False,  True],
 [ True,  True,  True, False],
 [False, False,  True,  True],
 [ True, False,  True, False]])
```

```
In [19]: # Where is True , it gives result , everything other that removed.we got value
```

```
G[G > 50]
```

```
Out[19]: array([64, 51, 75, 86, 53, 60, 95, 75, 79, 98, 87, 58, 56, 93])
```

it is best Techinque to filter the data in given condition

```
In [20]: # find out even numbers
```

```
G % 2 == 0
```

```
Out[20]: array([[ True, False, False,  True],
 [ True,  True,  True, False],
 [ True,  True, False, False],
 [False, False,  True,  True],
 [False, False, False,  True],
 [ True,  True, False, False]])
```

In [21]: # Gives only the even numbers

```
G [ G % 2 == 0]
```

Out[21]: array([64, 50, 8, 86, 6, 60, 50, 98, 34, 58, 56, 26])

In [22]: # find all numbers greater than 50 and are even

```
(G > 50 ) & (G % 2 == 0)
```

Out[22]: array([[ True, False, False, False],  
 [False, True, False, False],  
 [ True, False, False, False],  
 [False, False, True, False],  
 [False, False, False, True],  
 [ True, False, False, False]])

Here we used (&) bitwise Not logical(and) , because we are working with boolean values

In [23]: # Result

```
G [(G > 50 ) & (G % 2 == 0)]
```

Out[23]: array([64, 86, 60, 98, 58, 56])

In [24]: # find all numbers not divisible by 7

```
G % 7 == 0
```

Out[24]: array([[False, False, False, False],  
 [False, False, False, False],  
 [False, False, True, False],  
 [False, False, True, False],  
 [False, True, False, False],  
 [ True, False, False, False]])

In [25]: # Result

```
G[~(G % 7 == 0)] # (~) = Not
```

Out[25]: array([64, 51, 75, 50, 8, 86, 6, 53, 60, 50, 95, 75, 79, 34, 45, 87, 58,  
 26, 93, 17])

## Broadcasting

- Used in Vectorization

The term broadcasting describes how NumPy treats **arrays with different shapes during arithmetic operations.**

The smaller array is “broadcast” across the larger array so that they have compatible shapes.

In [26]: # same shape

```
a = np.arange(6).reshape(2,3)
b = np.arange(6,12).reshape(2,3)

print(a)
print(b)

print(a+b)
```

```
[[0 1 2]
 [3 4 5]]
 [[ 6  7  8]
 [ 9 10 11]]
 [[ 6  8 10]
 [12 14 16]]
```

In [27]: # diff shape

```
a = np.arange(6).reshape(2,3)
b = np.arange(3).reshape(1,3)

print(a)
print(b)

print(a+b)
```

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

## Broadcasting Rules

### 1. Make the two arrays have the same number of dimensions.

- If the numbers of dimensions of the two arrays are different, add new dimensions with size 1 to the head of the array with the smaller dimension.

**ex :** (3,2) = 2D , (3) =1D ---> Convert into (1,3)  
 (3,3,3) = 3D ,(3) = 1D ---> Convert into (1,1,3)

### 2. Make each dimension of the two arrays the same size.

- If the sizes of each dimension of the two arrays do not match, dimensions with size 1 are stretched to the size of the other array.

**ex :** (3,3)=2D ,(3) =1D ---> CONVERTED (1,3) than strech to (3,3)

- If there is a dimension whose size is not 1 in either of the two arrays, it cannot be broadcasted, and an error is raised.

$$\begin{array}{c}
 \text{(3,3)} \\
 \begin{array}{|c|c|c|} \hline 1 & 2 & 3 \\ \hline 4 & 5 & 6 \\ \hline 7 & 8 & 9 \\ \hline \end{array}
 \end{array}
 \begin{array}{c}
 \text{(3,) or (1,3)} \\
 \begin{array}{|c|c|c|} \hline -1 & 0 & 1 \\ \hline -1 & 0 & 1 \\ \hline -1 & 0 & 1 \\ \hline \end{array}
 \end{array}
 \begin{array}{c}
 \text{*} \\
 = \\
 \text{(3,3)}
 \end{array}
 \begin{array}{|c|c|c|} \hline -1 & 0 & 3 \\ \hline -4 & 0 & 6 \\ \hline -7 & 0 & 9 \\ \hline \end{array}
 \begin{array}{l}
 \text{multiplying several} \\
 \text{columns at once}
 \end{array}$$

$$\begin{array}{c}
 \text{(3,3)} \\
 \begin{array}{|c|c|c|} \hline 1 & 2 & 3 \\ \hline 4 & 5 & 6 \\ \hline 7 & 8 & 9 \\ \hline \end{array}
 \end{array}
 \begin{array}{c}
 \text{/} \\
 \text{(3,1)} \\
 \begin{array}{|c|c|c|} \hline 3 & 3 & 3 \\ \hline 6 & 6 & 6 \\ \hline 9 & 9 & 9 \\ \hline \end{array}
 \end{array}
 \begin{array}{c}
 = \\
 \text{(3,3)}
 \end{array}
 \begin{array}{|c|c|c|} \hline .3 & .7 & 1. \\ \hline .6 & .8 & 1. \\ \hline .8 & .9 & 1. \\ \hline \end{array}
 \begin{array}{l}
 \text{row-wise} \\
 \text{normalization}
 \end{array}$$

$$\begin{array}{c}
 \text{(3,) or (1,3)} \\
 \begin{array}{|c|c|c|} \hline 1 & 2 & 3 \\ \hline 1 & 2 & 3 \\ \hline 1 & 2 & 3 \\ \hline \end{array}
 \end{array}
 \begin{array}{c}
 \text{*} \\
 \text{(3,1)} \\
 \begin{array}{|c|c|c|} \hline 1 & 1 & 1 \\ \hline 2 & 2 & 2 \\ \hline 3 & 3 & 3 \\ \hline \end{array}
 \end{array}
 \begin{array}{c}
 = \\
 \text{(3,3)}
 \end{array}
 \begin{array}{|c|c|c|} \hline 1 & 2 & 3 \\ \hline 2 & 4 & 6 \\ \hline 3 & 6 & 9 \\ \hline \end{array}
 \begin{array}{l}
 \text{outer product}
 \end{array}$$

In [28]: # More examples

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

print(a) # 2 D
```

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

In [29]: print(b) # 1 D

```
[0 1 2]
```

In [30]: print(a+b) # Arthematic Operation

```
[[ 0  2  4]
 [ 3  5  7]
 [ 6  8 10]
 [ 9 11 13]]
```

EXPLANATION : Arthematic Operation possible because , Here a = (4,3) is 2D and b =(3) is 1D so did converted (3) to (1,3) and streched to (4,3)

In [31]: # Could not Broadcast

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

print(a)
print(b)

print(a+b)
```

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

---

```
-----  
ValueError                                                 Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_9360/470058718.py in <module>
      7 print(b)
      8
----> 9 print(a+b)
```

**ValueError:** operands could not be broadcast together with shapes (3,4) (3,)

**EXPLANATION :** Arthematic Operation **not** possible because , Here a = (3,4) is 2D and b =(3) is 1D so did converted (3) to (1,3) and streched to (3,3) but , a is not equals to b . so it got failed

In [32]: a = np.arange(3).reshape(1,3)
b = np.arange(3).reshape(3,1)

```
print(a)
print(b)

print(a+b)
```

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

**EXPLANATION :** Arthematic Operation possible because , Here a = (1,3) is 2D and b =(3,1) is 2D so did converted (1,3) to (3,3) and b(3,1) convert (1)to 3 than (3,3) . finally it equally.

```
In [33]: a = np.arange(3).reshape(1,3)
b = np.arange(4).reshape(4,1)

print(a)
print(b)

print(a + b)
```

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

EXPLANATION : Same as before

```
In [34]: a = np.array([1])
# shape -> (1,1) stretched to 2,2
b = np.arange(4).reshape(2,2)
# shape -> (2,2)

print(a)
print(b)

print(a+b)
```

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

In [35]: # doesn't work

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

print(a)
print(b)

print(a+b)
```

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

---

**ValueError**

Traceback (most recent call last)

```
~\AppData\Local\Temp\ipykernel_9360/1200695402.py in <module>
      7 print(b)
      8
----> 9 print(a+b)
```

**ValueError:** operands could not be broadcast together with shapes (3,4) (4,3)

**EXPLANATION :** there is no 1 to convert ,so got failed

In [36]: # Not Work

```
a = np.arange(16).reshape(4,4)
b = np.arange(4).reshape(2,2)

print(a)
print(b)

print(a+b)
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]
 [12 13 14 15]]
[[0 1]
 [2 3]]
```

---

**ValueError**

Traceback (most recent call last)

```
~\AppData\Local\Temp\ipykernel_9360/2417388683.py in <module>
      6 print(b)
      7
----> 8 print(a+b)
```

**ValueError:** operands could not be broadcast together with shapes (4,4) (2,2)

**EXPLANATION :** there is no 1 to convert ,so got failed

## Working with mathematical formulas

```
In [37]: k = np.arange(10)
```

```
In [38]: k
```

```
Out[38]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
In [39]: np.sum(k)
```

```
Out[39]: 45
```

```
In [40]: np.sin(k)
```

```
Out[40]: array([ 0.           ,  0.84147098,  0.90929743,  0.14112001, -0.7568025 ,
 -0.95892427, -0.2794155 ,  0.6569866 ,  0.98935825,  0.41211849])
```

### sigmoid

```
In [44]: def sigmoid(array):
    return 1/(1+np.exp(-(array)))
k = np.arange(10)
sigmoid(k)
```

```
Out[44]: array([0.5           , 0.73105858, 0.88079708, 0.95257413, 0.98201379,
 0.99330715, 0.99752738, 0.99908895, 0.99966465, 0.99987661])
```

```
In [45]: k = np.arange(100)
sigmoid(k)
```

## mean squared error

```
In [46]: actual = np.random.randint(1,50,25)
predicted = np.random.randint(1,50,25)
```

In [47]: actual

**Out[47]:** array([17, 4, 4, 24, 18, 44, 22, 25, 17, 39, 3, 34, 37, 12, 47, 22, 37, 9, 47, 38, 27, 46, 47, 34, 8])

```
In [48]: predicted
```

```
Out[48]: array([47, 31, 30, 17, 7, 22, 1, 16, 1, 24, 16, 7, 6, 37, 18, 15, 2, 33, 25, 33, 9, 17, 36, 7, 16])
```

```
In [50]: def mse(actual,predicted):
    return np.mean((actual-predicted)**2)

mse(actual,predicted)
```

Out[50]: 469.0

```
In [51]: # detailed
```

## actual-predicted

```
Out[51]: array([-30, -27, -26, 7, 11, 22, 21, 9, 16, 15, -13, 27, 31,
-25, 29, 7, 35, -24, 22, 5, 18, 29, 11, 27, -8])
```

```
In [52]: (actual-predicted)**2
```

```
Out[52]: array([ 900,  729,  676,   49,  121,  484,  441,   81,  256,  225,  169,
    729,  961,  625,  841,   49, 1225,  576,  484,   25,  324,  841,
   121,  729,   64], dtype=int32)
```

```
In [53]: np.mean((actual-predicted)**2)
```

```
Out[53]: 469.0
```

## Working with Missing Values

```
In [55]: # Working with missing values -> np.nan
```

```
S = np.array([1,2,3,4,np.nan,6])
S
```

```
Out[55]: array([ 1.,  2.,  3.,  4., nan,  6.])
```

```
In [56]: np.isnan(S)
```

```
Out[56]: array([False, False, False, False,  True, False])
```

```
In [57]: S[np.isnan(S)] # Nan values
```

```
Out[57]: array([nan])
```

```
In [58]: S[~np.isnan(S)] # Not Nan Values
```

```
Out[58]: array([1., 2., 3., 4., 6.])
```

## Plotting Graphs

In [59]: # plotting a 2D plot

```
# x = y
```

```
x = np.linspace(-10,10,100)
```

```
x
```

Out[59]: array([-10. , -9.7979798 , -9.5959596 , -9.39393939,

```
-9.19191919, -8.98989899, -8.78787879, -8.58585859,
```

```
-8.38383838, -8.18181818, -7.97979798, -7.77777778,
```

```
-7.57575758, -7.37373737, -7.17171717, -6.96969697,
```

```
-6.76767677, -6.56565657, -6.36363636, -6.16161616,
```

```
-5.95959596, -5.75757576, -5.55555556, -5.35353535,
```

```
-5.15151515, -4.94949495, -4.74747475, -4.54545455,
```

```
-4.34343434, -4.14141414, -3.93939394, -3.73737374,
```

```
-3.53535354, -3.33333333, -3.13131313, -2.92929293,
```

```
-2.72727273, -2.52525253, -2.32323232, -2.12121212,
```

```
-1.91919192, -1.71717172, -1.51515152, -1.31313131,
```

```
-1.11111111, -0.90909091, -0.70707071, -0.50505051,
```

```
-0.3030303 , -0.1010101 , 0.1010101 , 0.3030303 ,
```

```
0.50505051, 0.70707071, 0.90909091, 1.11111111,
```

```
1.31313131, 1.51515152, 1.71717172, 1.91919192,
```

```
2.12121212, 2.32323232, 2.52525253, 2.72727273,
```

```
2.92929293, 3.13131313, 3.33333333, 3.53535354,
```

```
3.73737374, 3.93939394, 4.14141414, 4.34343434,
```

```
4.54545455, 4.74747475, 4.94949495, 5.15151515,
```

```
5.35353535, 5.55555556, 5.75757576, 5.95959596,
```

```
6.16161616, 6.36363636, 6.56565657, 6.76767677,
```

```
6.96969697, 7.17171717, 7.37373737, 7.57575758,
```

```
7.77777778, 7.97979798, 8.18181818, 8.38383838,
```

```
8.58585859, 8.78787879, 8.98989899, 9.19191919,
```

```
9.39393939, 9.5959596 , 9.7979798 , 10. ])
```

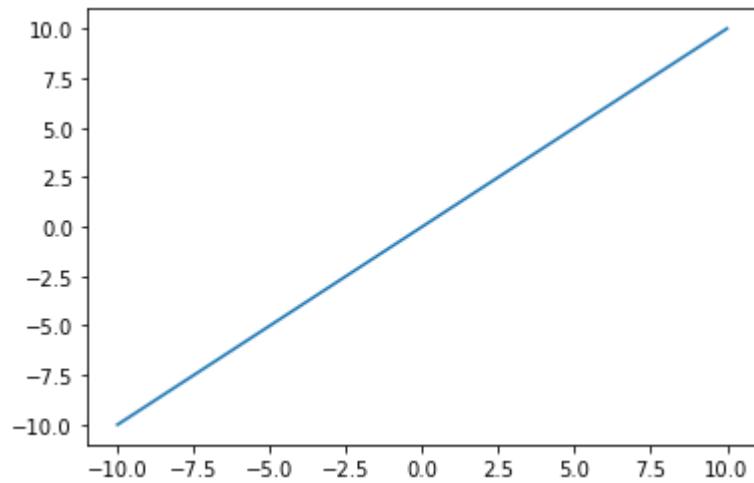
In [60]: y = x

In [61]: `y`

```
Out[61]: array([-10.        , -9.7979798 , -9.5959596 , -9.39393939,
   -9.19191919, -8.98989899, -8.78787879, -8.58585859,
   -8.38383838, -8.18181818, -7.97979798, -7.77777778,
   -7.57575758, -7.37373737, -7.17171717, -6.96969697,
   -6.76767677, -6.56565657, -6.36363636, -6.16161616,
   -5.95959596, -5.75757576, -5.55555556, -5.35353535,
   -5.15151515, -4.94949495, -4.74747475, -4.54545455,
   -4.34343434, -4.14141414, -3.93939394, -3.73737374,
   -3.53535354, -3.33333333, -3.13131313, -2.92929293,
   -2.72727273, -2.52525253, -2.32323232, -2.12121212,
   -1.91919192, -1.71717172, -1.51515152, -1.31313131,
   -1.11111111, -0.90909091, -0.70707071, -0.50505051,
   -0.3030303 , -0.1010101 ,  0.1010101 ,  0.3030303 ,
   0.50505051,  0.70707071,  0.90909091,  1.11111111,
   1.31313131,  1.51515152,  1.71717172,  1.91919192,
   2.12121212,  2.32323232,  2.52525253,  2.72727273,
   2.92929293,  3.13131313,  3.33333333,  3.53535354,
   3.73737374,  3.93939394,  4.14141414,  4.34343434,
   4.54545455,  4.74747475,  4.94949495,  5.15151515,
   5.35353535,  5.55555556,  5.75757576,  5.95959596,
   6.16161616,  6.36363636,  6.56565657,  6.76767677,
   6.96969697,  7.17171717,  7.37373737,  7.57575758,
   7.77777778,  7.97979798,  8.18181818,  8.38383838,
   8.58585859,  8.78787879,  8.98989899,  9.19191919,
   9.39393939,  9.5959596 ,  9.7979798 ,  10.        ])
```

In [62]: `import matplotlib.pyplot as plt`  
`plt.plot(x ,y)`

```
Out[62]: [<matplotlib.lines.Line2D at 0x1172fe48bb0>]
```

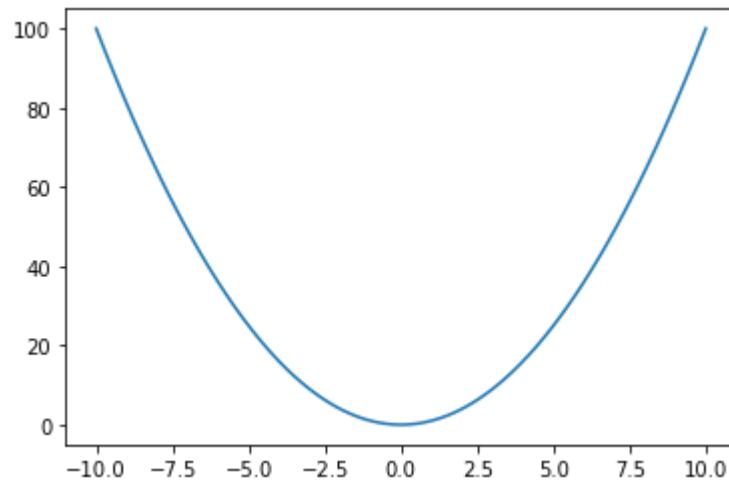


In [63]: #  $y = x^2$

```
x = np.linspace(-10,10,100)
y = x**2

plt.plot(x,y)
```

Out[63]: [`<matplotlib.lines.Line2D at 0x117324e7310>`]

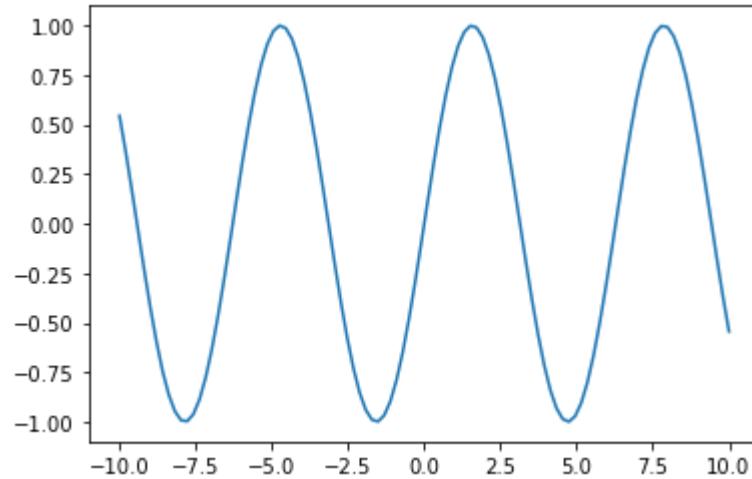


In [64]: #  $y = \sin(x)$

```
x = np.linspace(-10,10,100)
y = np.sin(x)

plt.plot(x,y)
```

Out[64]: [`<matplotlib.lines.Line2D at 0x11732560190>`]

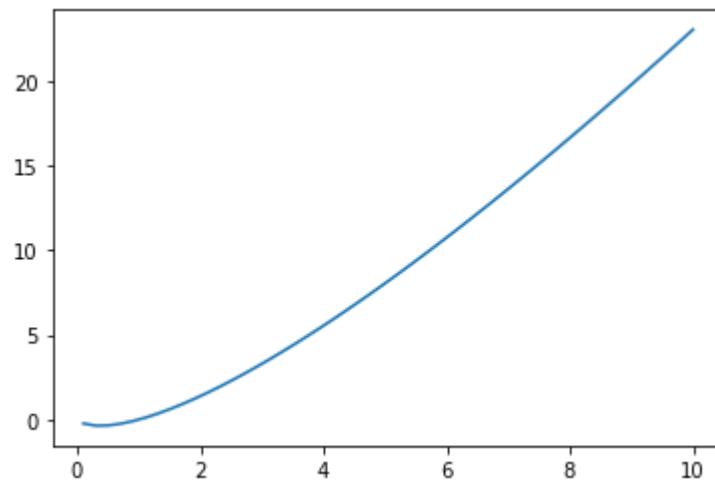


```
In [65]: # y = xLog(x)
x = np.linspace(-10,10,100)
y = x * np.log(x)

plt.plot(x,y)
```

C:\Users\user\AppData\Local\Temp\ipykernel\_9360/2564014901.py:3: RuntimeWarning: invalid value encountered in log  
y = x \* np.log(x)

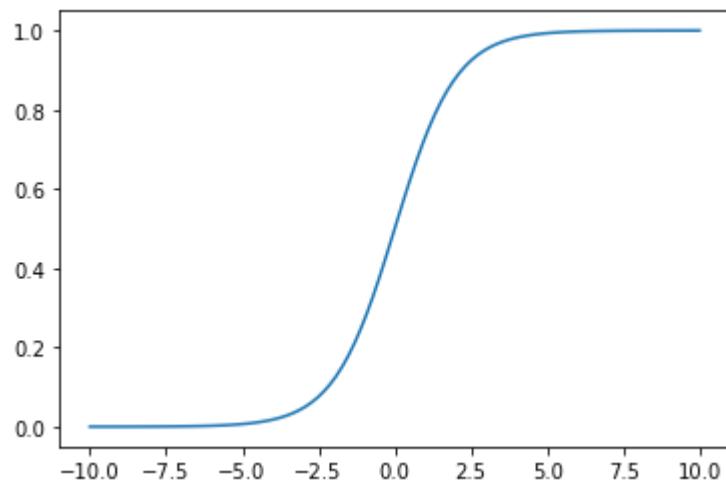
Out[65]: [`<matplotlib.lines.Line2D at 0x117325c97f0>`]



```
In [66]: # sigmoid
x = np.linspace(-10,10,100)
y = 1/(1+np.exp(-x))

plt.plot(x,y)
```

Out[66]: [`<matplotlib.lines.Line2D at 0x1173262f700>`]



In [ ]:

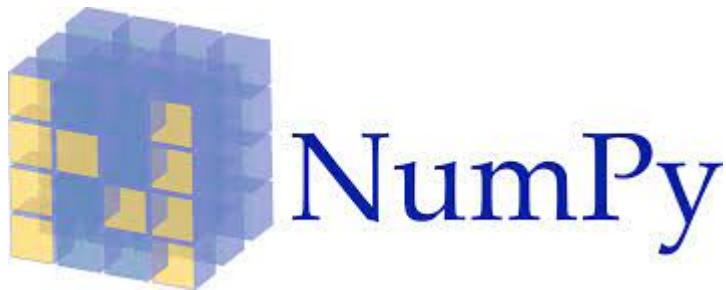


```
In [1]: import numpy as np
import matplotlib.pyplot as plt
```

## Meshgrid

Meshgrids are a way to **create coordinate matrices from coordinate vectors**. In NumPy,

- the meshgrid function is used to generate a coordinate grid given 1D coordinate arrays. It produces two 2D arrays representing the x and y coordinates of each point on the grid



The **np.meshgrid** function is used primarily for

- Creating/Plotting 2D functions  $f(x,y)$
- Generating combinations of 2 or more numbers

Example: How you might think to create a 2D function  $f(x,y)$

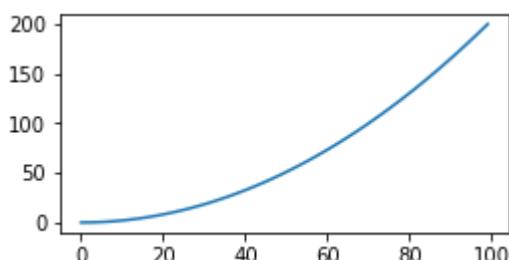
```
In [2]: x = np.linspace(0,10,100)
y = np.linspace(0,10,100)
```

Try to create 2D function

```
In [3]: f = x**2+y**2
```

Plot

```
In [4]: plt.figure(figsize=(4,2))
plt.plot(f)
plt.show()
```



But f is a 1 dimensional function! How does one generate a surface plot?

```
In [5]: x = np.arange(3)
y = np.arange(3)
```

```
In [6]: x
```

```
Out[6]: array([0, 1, 2])
```

```
In [7]: y
```

```
Out[7]: array([0, 1, 2])
```

Generating a meshgrid:

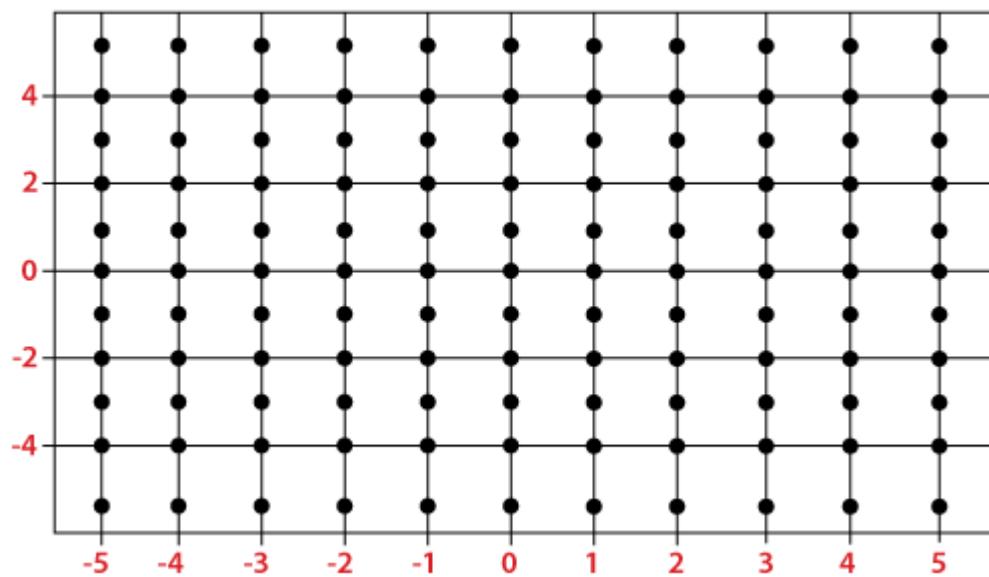
```
In [8]: xv ,yv = np.meshgrid(x,y)
```

```
In [9]: xv
```

```
Out[9]: array([[0, 1, 2],
               [0, 1, 2],
               [0, 1, 2]])
```

```
In [10]: yv
```

```
Out[10]: array([[0, 0, 0],
                 [1, 1, 1],
                 [2, 2, 2]])
```



```
In [11]: P = np.linspace(-4, 4, 9)
V = np.linspace(-5, 5, 11)
print(P)
print(V)
```

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

```
In [12]: P_1, V_1 = np.meshgrid(P,V)
```

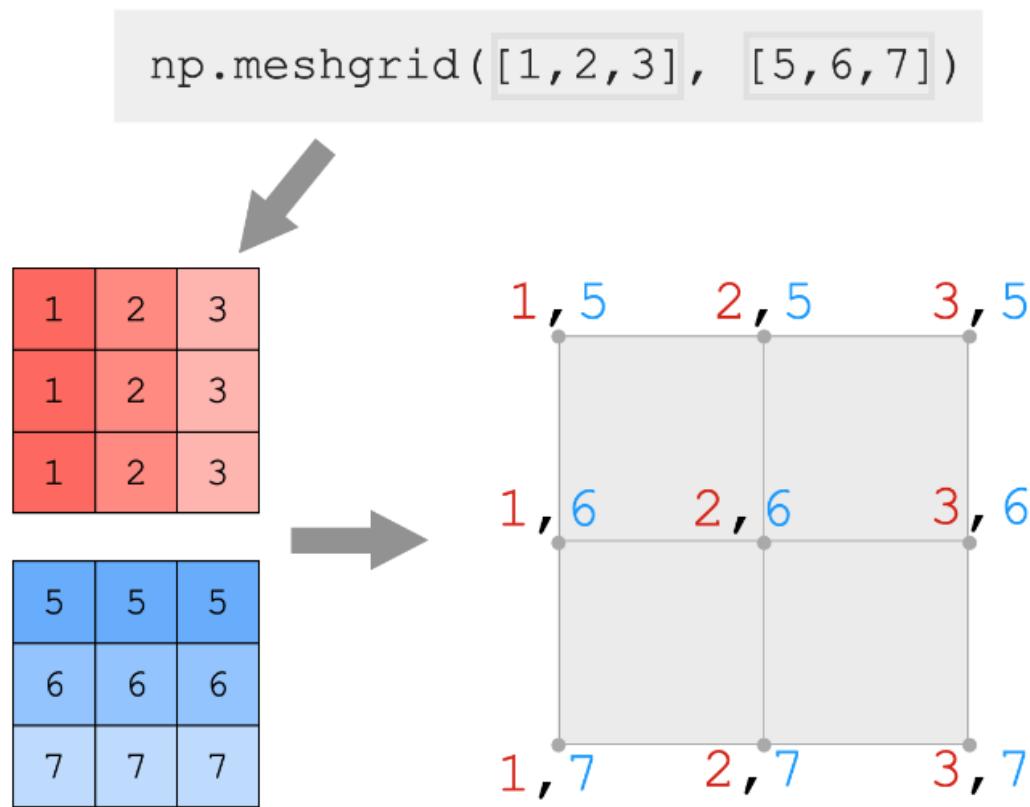
```
In [13]: print(P_1)
```

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

```
In [14]: print(V_1)
```

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

## Numpy Meshgrid Creates Coordinates for a Grid System



These arrays, xv and yv, each separately give the x and y coordinates on a 2D grid. You can do normal numpy operations on these arrays:

```
In [15]: xv**2 + yv**2
```

```
Out[15]: array([[0, 1, 4],
 [1, 2, 5],
 [4, 5, 8]], dtype=int32)
```

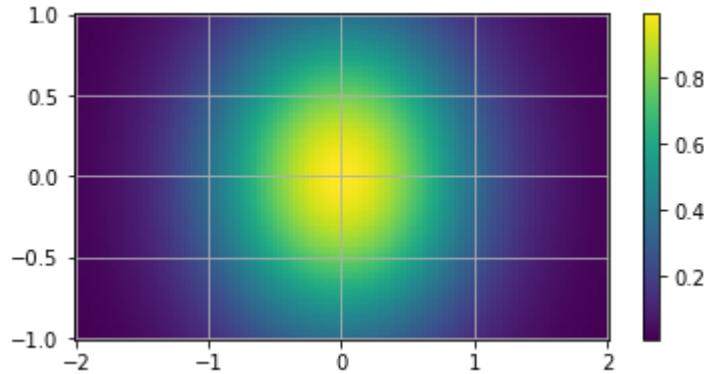
This can be done on a larger scale to plot surface plots of 2D functions

Generate functions  $f(x, y) = e^{-(x^2+y^2)}$  for  $-2 \leq x \leq 2$  and  $-1 \leq y \leq 1$

```
In [16]: x = np.linspace(-2,2,100)
y = np.linspace(-1,1,100)
xv, yv = np.meshgrid(x, y)
f = np.exp(-xv**2-yv**2)
```

Note: pcolormesh is typically the preferable function for 2D plotting, as opposed to imshow or pcolor, which take longer.)

```
In [17]: plt.figure(figsize=(6, 3))
plt.pcolormesh(xv, yv, f, shading='auto')
plt.colorbar()
plt.grid()
plt.show()
```



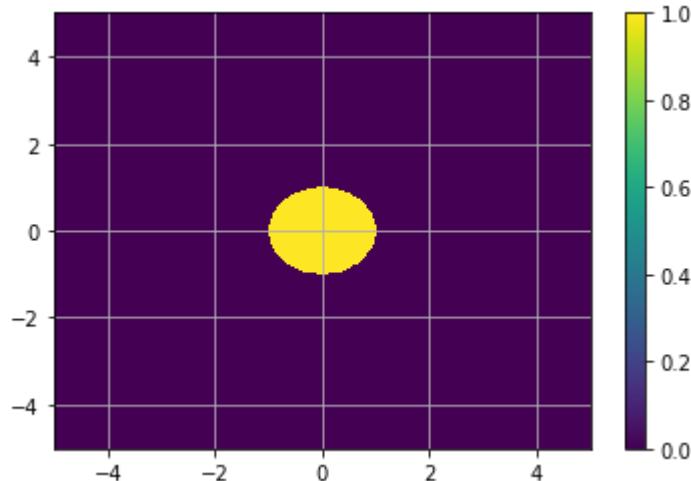
$$f(x,y) = 1 \text{ & } x^2+y^2 < 1 \setminus 0 \text{ & } x^2+y^2 > 1$$

```
In [18]: import numpy as np
import matplotlib.pyplot as plt

def f(x, y):
    return np.where((x**2 + y**2 < 1), 1.0, 0.0)

x = np.linspace(-5, 5, 500)
y = np.linspace(-5, 5, 500)
xv, yv = np.meshgrid(x, y)
rectangular_mask = f(xv, yv)

plt.pcolormesh(xv, yv, rectangular_mask, shading='auto')
plt.colorbar()
plt.grid()
plt.show()
```



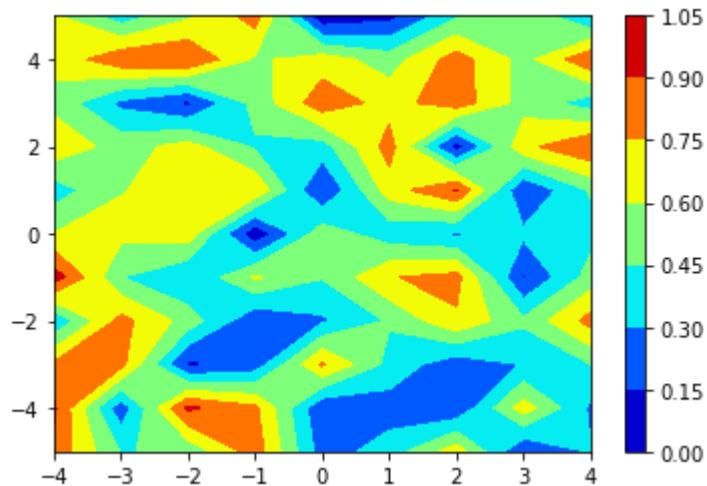
```
In [19]: # numpy.linspace creates an array of  
# 9 linearly placed elements between  
# -4 and 4, both inclusive  
  
x = np.linspace(-4, 4, 9)
```

```
In [20]: # numpy.linspace creates an array of  
# 9 linearly placed elements between  
# -4 and 4, both inclusive
```

```
In [21]: y = np.linspace(-5, 5, 11)
```

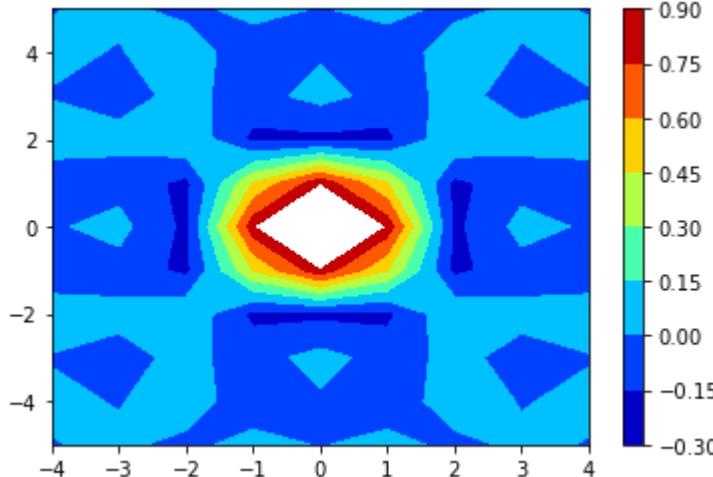
```
In [22]: x_1, y_1 = np.meshgrid(x, y)
```

```
In [23]: random_data = np.random.random((11, 9))  
plt.contourf(x_1, y_1, random_data, cmap = 'jet')  
  
plt.colorbar()  
plt.show()
```



```
In [24]: sine = (np.sin(x_1**2 + y_1**2))/(x_1**2 + y_1**2)
plt.contourf(x_1, y_1, sine, cmap = 'jet')
plt.colorbar()
plt.show()
```

C:\Users\user\AppData\Local\Temp\ipykernel\_3612\3873722910.py:1: RuntimeWarning: invalid value encountered in true\_divide  
 sine = (np.sin(x\_1\*\*2 + y\_1\*\*2))/(x\_1\*\*2 + y\_1\*\*2)



We observe that  $x_1$  is a row repeated matrix whereas  $y_1$  is a column repeated matrix. One row of  $x_1$  and one column of  $y_1$  is enough to determine the positions of all the points as the other values will get repeated over and over.

```
In [25]: x_1, y_1 = np.meshgrid(x, y, sparse = True)
```

```
In [26]: x_1
```

```
Out[26]: array([[-4., -3., -2., -1.,  0.,  1.,  2.,  3.,  4.]])
```

```
In [27]: y_1
```

```
Out[27]: array([[ -5.],
   [ -4.],
   [ -3.],
   [ -2.],
   [ -1.],
   [  0.],
   [  1.],
   [  2.],
   [  3.],
   [  4.],
   [  5.]])
```

The shape of `x_1` changed from (11, 9) to (1, 9) and that of `y_1` changed from (11, 9) to (11, 1).  
The indexing of Matrix is however different. Actually, it is the exact opposite of Cartesian indexing.

## np.sort

Return a sorted copy of an array.

```
In [28]: a = np.random.randint(1,100,15) #1D
a
```

```
Out[28]: array([46, 53, 15, 44, 33, 39, 76, 60, 68, 12, 87, 66, 74, 10, 98])
```

```
In [31]: b = np.random.randint(1,100,24).reshape(6,4) # 2D
b
```

```
Out[31]: array([[ 6, 51, 40, 85],
 [35, 28, 91, 68],
 [27, 30, 6, 4],
 [18, 48, 48, 15],
 [35, 45, 99, 17],
 [42, 29, 88, 31]])
```

```
In [32]: np.sort(a) # Default= Ascending
```

```
Out[32]: array([10, 12, 15, 33, 39, 44, 46, 53, 60, 66, 68, 74, 76, 87, 98])
```

```
In [36]: np.sort(a)[::-1] # Descending order
```

```
Out[36]: array([98, 87, 76, 74, 68, 66, 60, 53, 46, 44, 39, 33, 15, 12, 10])
```

```
In [33]: np.sort(b) # row rise sorting
```

```
Out[33]: array([[ 6, 40, 51, 85],
 [28, 35, 68, 91],
 [ 4, 6, 27, 30],
 [15, 18, 48, 48],
 [17, 35, 45, 99],
 [29, 31, 42, 88]])
```

```
In [35]: np.sort(b,axis = 0) # column rise sorting
```

```
Out[35]: array([[ 6, 28,  6,  4],
 [18, 29, 40, 15],
 [27, 30, 48, 17],
 [35, 45, 88, 31],
 [35, 48, 91, 68],
 [42, 51, 99, 85]])
```

## np.append

The numpy.append() appends values along the mentioned axis at the end of the array

In [37]: # code

```
a
```

Out[37]: array([46, 53, 15, 44, 33, 39, 76, 60, 68, 12, 87, 66, 74, 10, 98])

In [38]: np.append(a, 200)

Out[38]: array([ 46, 53, 15, 44, 33, 39, 76, 60, 68, 12, 87, 66, 74, 10, 98, 200])

In [39]: b # on 2D

Out[39]: array([[ 6, 51, 40, 85],  
[ 35, 28, 91, 68],  
[ 27, 30, 6, 4],  
[ 18, 48, 48, 15],  
[ 35, 45, 99, 17],  
[ 42, 29, 88, 31]])

In [42]: # Adding Extra column :1

```
np.append(b,np.ones((b.shape[0],1)))
```

Out[42]: array([ 6., 51., 40., 85., 35., 28., 91., 68., 27., 30., 6., 4., 18.,  
48., 48., 15., 35., 45., 99., 17., 42., 29., 88., 31., 1., 1.,  
1., 1., 1., 1.])

In [43]: np.append(b,np.ones((b.shape[0],1)),axis=1)

Out[43]: array([[ 6., 51., 40., 85., 1.],  
[ 35., 28., 91., 68., 1.],  
[ 27., 30., 6., 4., 1.],  
[ 18., 48., 48., 15., 1.],  
[ 35., 45., 99., 17., 1.],  
[ 42., 29., 88., 31., 1.]])

In [44]: #Adding random numbers in new column

```
np.append(b,np.random.random((b.shape[0],1)),axis=1)
```

Out[44]: array([[ 6. , 51. , 40. , 85. , 0.47836639],  
[ 35. , 28. , 91. , 68. , 0.98776768],  
[ 27. , 30. , 6. , 4. , 0.55833259],  
[ 18. , 48. , 48. , 15. , 0.7730807 ],  
[ 35. , 45. , 99. , 17. , 0.22512908],  
[ 42. , 29. , 88. , 31. , 0.73795824]])

## np.concatenate

numpy.concatenate() function concatenate a sequence of arrays along an existing axis.

```
In [45]: # code
c = np.arange(6).reshape(2,3)
d = np.arange(6,12).reshape(2,3)
```

```
In [46]: c
```

```
Out[46]: array([[0, 1, 2],
 [3, 4, 5]])
```

```
In [47]: d
```

```
Out[47]: array([[ 6,  7,  8],
 [ 9, 10, 11]])
```

we can use it replacement of **vstack** and **hstack**

```
In [48]: np.concatenate((c,d)) # Row wise
```

```
Out[48]: array([[ 0,  1,  2],
 [ 3,  4,  5],
 [ 6,  7,  8],
 [ 9, 10, 11]])
```

```
In [49]: np.concatenate((c,d),axis =1 ) # column wise
```

```
Out[49]: array([[ 0,  1,  2,  6,  7,  8],
 [ 3,  4,  5,  9, 10, 11]])
```

## np.unique

With the help of np.unique() method, we can get the unique values from an array given as parameter in np.unique() method.

```
In [50]: # code
e = np.array([1,1,2,2,3,3,4,4,5,5,6,6])
```

```
In [51]: e
```

```
Out[51]: array([1, 1, 2, 2, 3, 3, 4, 4, 5, 5, 6, 6])
```

```
In [52]: np.unique(e)
```

```
Out[52]: array([1, 2, 3, 4, 5, 6])
```

## np.expand\_dims

With the help of Numpy.expand\_dims() method, we can get the expanded **dimensions of an array**

In [53]: #code

```
a
```

Out[53]: array([46, 53, 15, 44, 33, 39, 76, 60, 68, 12, 87, 66, 74, 10, 98])

In [57]: a.shape # 1 D

Out[57]: (15,)

In [56]: # converting into 2D array

```
np.expand_dims(a, axis = 0)
```

Out[56]: array([[46, 53, 15, 44, 33, 39, 76, 60, 68, 12, 87, 66, 74, 10, 98]])

In [59]: np.expand\_dims(a, axis = 0).shape # 2D

Out[59]: (1, 15)

In [60]: np.expand\_dims(a, axis = 1)

**Download Machine Learning Study Material:**

<https://t.me/AIMLDeepThaught>

Out[60]: array([[46],  
[53],  
[15],  
[44],  
[33],  
[39],  
[76],  
[60],  
[68],  
[12],  
[87],  
[66],  
[74],  
[10],  
[98]])

We can use in row vector and Column vector .

expand\_dims() is used to **insert an addition dimension in input Tensor.**

In [61]: np.expand\_dims(a, axis = 1).shape

Out[61]: (15, 1)

## np.where

The numpy.where() function returns the indices of elements in an input array where the given condition is satisfied.

In [62]:

```
a
```

Out[62]:

```
array([46, 53, 15, 44, 33, 39, 76, 60, 68, 12, 87, 66, 74, 10, 98])
```

In [63]:

```
# find all indices with value greater than 50
```

```
np.where(a>50)
```

Out[63]:

```
(array([ 1,  6,  7,  8, 10, 11, 12, 14], dtype=int64),)
```

```
np.where( condition, True , false)
```

In [64]:

```
# replace all values > 50 with 0
```

```
np.where(a>50,0,a)
```

Out[64]:

```
array([46, 0, 15, 44, 33, 39, 0, 0, 0, 12, 0, 0, 0, 10, 0])
```

In [67]:

```
# print and replace all even numbers to 0
```

```
np.where(a%2 == 0,0,a)
```

Out[67]:

```
array([ 0, 53, 15, 0, 33, 39, 0, 0, 0, 0, 87, 0, 0, 0, 0])
```

## np.argmax

The numpy.argmax() function returns **indices of the max element of the array in a particular axis.**

**arg** = argument

In [68]:

```
# code
```

```
a
```

Out[68]:

```
array([46, 53, 15, 44, 33, 39, 76, 60, 68, 12, 87, 66, 74, 10, 98])
```

In [69]:

```
np.argmax(a) # biggest number : index number
```

Out[69]:

```
14
```

In [71]: b # on 2D

```
Out[71]: array([[ 6, 51, 40, 85],
   [35, 28, 91, 68],
   [27, 30,  6,  4],
   [18, 48, 48, 15],
   [35, 45, 99, 17],
   [42, 29, 88, 31]])
```

In [72]: np.argmax(b, axis =1) # row wise biggest number : index

```
Out[72]: array([3, 2, 1, 1, 2, 2], dtype=int64)
```

In [73]: np.argmax(b, axis =0) # column wise biggest number : index

```
Out[73]: array([5, 0, 4, 0], dtype=int64)
```

In [75]: # np.argmin

```
a
```

```
Out[75]: array([46, 53, 15, 44, 33, 39, 76, 60, 68, 12, 87, 66, 74, 10, 98])
```

In [76]: np.argmin(a)

```
Out[76]: 13
```

## On Statistics:

### np.cumsum

numpy.cumsum() function is used when we want to compute the **cumulative sum** of array elements over a given axis.

In [77]: a

```
Out[77]: array([46, 53, 15, 44, 33, 39, 76, 60, 68, 12, 87, 66, 74, 10, 98])
```

In [79]: np.cumsum(a)

```
Out[79]: array([ 46,  99, 114, 158, 191, 230, 306, 366, 434, 446, 533, 599, 673,
   683, 781], dtype=int32)
```

In [85]: b

```
Out[85]: array([[ 6, 51, 40, 85],
 [35, 28, 91, 68],
 [27, 30, 6, 4],
 [18, 48, 48, 15],
 [35, 45, 99, 17],
 [42, 29, 88, 31]])
```

In [86]: np.cumsum(b)

```
Out[86]: array([ 6, 57, 97, 182, 217, 245, 336, 404, 431, 461, 467, 471, 489,
 537, 585, 600, 635, 680, 779, 796, 838, 867, 955, 986], dtype=int32)
```

In [84]: np.cumsum(b, axis=1) # row wise calculation or cumulative sum

```
Out[84]: array([[ 6, 57, 97, 182],
 [35, 63, 154, 222],
 [27, 57, 63, 67],
 [18, 66, 114, 129],
 [35, 80, 179, 196],
 [42, 71, 159, 190]], dtype=int32)
```

In [87]: np.cumsum(b, axis=0) # column wise calculation or cumulative sum

```
Out[87]: array([[ 6, 51, 40, 85],
 [41, 79, 131, 153],
 [68, 109, 137, 157],
 [86, 157, 185, 172],
 [121, 202, 284, 189],
 [163, 231, 372, 220]], dtype=int32)
```

In [88]: # np.cumprod --> Multiply

a

```
Out[88]: array([46, 53, 15, 44, 33, 39, 76, 60, 68, 12, 87, 66, 74, 10, 98])
```

In [89]: np.cumprod(a)

```
Out[89]: array([ 46, 2438, 36570, 1609080, 53099640,
 2070885960, -1526456992, -1393106304, -241948160, 1391589376,
 809191424, 1867026432, 721002496, -1379909632, -2087157760],
 dtype=int32)
```

## np.percentile

numpy.percentile() function used to compute the **n<sup>th</sup> percentile** of the given data (array elements) along the specified axis.

In [90]: a

Out[90]: array([46, 53, 15, 44, 33, 39, 76, 60, 68, 12, 87, 66, 74, 10, 98])

In [91]: np.percentile(a, 100) # Max

Out[91]: 98.0

In [92]: np.percentile(a, 0) # Min

Out[92]: 10.0

In [93]: np.percentile(a, 50) # Median

Out[93]: 53.0

In [94]: np.median(a)

Out[94]: 53.0

## np.histogram

Numpy has a built-in numpy.histogram() function which represents the **frequency of data distribution** in the graphical form.

In [95]: a

Out[95]: array([46, 53, 15, 44, 33, 39, 76, 60, 68, 12, 87, 66, 74, 10, 98])

In [98]: np.histogram(a, bins=[10, 20, 30, 40, 50, 60, 70, 80, 90, 100])

Out[98]: (array([3, 0, 2, 2, 1, 3, 2, 1, 1], dtype=int64),  
 array([ 10, 20, 30, 40, 50, 60, 70, 80, 90, 100]))

In [99]: np.histogram(a, bins=[0, 50, 100])

Out[99]: (array([7, 8], dtype=int64), array([ 0, 50, 100]))

## np.corrcoef

Return Pearson product-moment correlation coefficients.

In [101]: salary = np.array([20000, 40000, 25000, 35000, 60000])  
experience = np.array([1, 3, 2, 4, 2])

```
In [102]: salary
```

```
Out[102]: array([20000, 40000, 25000, 35000, 60000])
```

```
In [103]: experience
```

```
Out[103]: array([1, 3, 2, 4, 2])
```

```
In [104]: np.corrcoef(salary,experience) # Correlation Coefficient
```

```
Out[104]: array([[1.          , 0.25344572],
                  [0.25344572, 1.         ]])
```

## Utility functions

### np.isin

With the help of numpy.isin() method, we can see that one array having values are checked in a different numpy array having different elements with different sizes.

```
In [105]: # code
```

```
a
```

```
Out[105]: array([46, 53, 15, 44, 33, 39, 76, 60, 68, 12, 87, 66, 74, 10, 98])
```

```
In [107]: items = [10,20,30,40,50,60,70,80,90,100]
```

```
np.isin(a,items)
```

```
Out[107]: array([False, False, False, False, False, False, False, True, False,
                  False, False, False, True, False])
```

```
In [108]: a[np.isin(a,items)]
```

```
Out[108]: array([60, 10])
```

### np.flip

The numpy.flip() function **reverses the order** of array elements along the specified axis, preserving the shape of the array.

```
In [109]: # code
```

```
a
```

```
Out[109]: array([46, 53, 15, 44, 33, 39, 76, 60, 68, 12, 87, 66, 74, 10, 98])
```

In [110]: `np.flip(a) # reverse`

Out[110]: `array([98, 10, 74, 66, 87, 12, 68, 60, 76, 39, 33, 44, 15, 53, 46])`

In [111]: `b`

Out[111]: `array([[ 6, 51, 40, 85],  
[ 35, 28, 91, 68],  
[ 27, 30, 6, 4],  
[ 18, 48, 48, 15],  
[ 35, 45, 99, 17],  
[ 42, 29, 88, 31]])`

In [112]: `np.flip(b)`

Out[112]: `array([[ 31, 88, 29, 42],  
[ 17, 99, 45, 35],  
[ 15, 48, 48, 18],  
[ 4, 6, 30, 27],  
[ 68, 91, 28, 35],  
[ 85, 40, 51, 6]])`

In [113]: `np.flip(b, axis = 1) # row`

Out[113]: `array([[ 85, 40, 51, 6],  
[ 68, 91, 28, 35],  
[ 4, 6, 30, 27],  
[ 15, 48, 48, 18],  
[ 17, 99, 45, 35],  
[ 31, 88, 29, 42]])`

In [114]: `np.flip(b, axis = 0) # column`

Out[114]: `array([[ 42, 29, 88, 31],  
[ 35, 45, 99, 17],  
[ 18, 48, 48, 15],  
[ 27, 30, 6, 4],  
[ 35, 28, 91, 68],  
[ 6, 51, 40, 85]])`

## np.put

The numpy.put() function **replaces** specific elements of an array with given values of p\_array.  
Array indexed works on flattened array.

In [115]: `# code`

`a`

Out[115]: `array([46, 53, 15, 44, 33, 39, 76, 60, 68, 12, 87, 66, 74, 10, 98])`

```
In [116]: np.put(a,[0,1],[110,530]) # permanent changes
```

```
In [117]: a
```

```
Out[117]: array([110, 530, 15, 44, 33, 39, 76, 60, 68, 12, 87, 66, 74,
10, 98])
```

## np.delete

The numpy.delete() function returns a new array with the deletion of sub-arrays along with the mentioned axis.

```
In [118]: # code
```

```
a
```

```
Out[118]: array([110, 530, 15, 44, 33, 39, 76, 60, 68, 12, 87, 66, 74,
10, 98])
```

```
In [119]: np.delete(a,0) # deleted 0 index item
```

```
Out[119]: array([530, 15, 44, 33, 39, 76, 60, 68, 12, 87, 66, 74, 10,
98])
```

```
In [120]: np.delete(a,[0,2,4]) # deleted 0,2,4 index items
```

```
Out[120]: array([530, 44, 39, 76, 60, 68, 12, 87, 66, 74, 10, 98])
```

## Set functions

- np.union1d
- np.intersect1d
- np.setdiff1d
- np.setxor1d
- np.in1d

```
In [121]: m = np.array([1,2,3,4,5])
n = np.array([3,4,5,6,7])
```

```
In [122]: # Union
```

```
np.union1d(m,n)
```

```
Out[122]: array([1, 2, 3, 4, 5, 6, 7])
```

In [123]: # Intersection

```
np.intersect1d(m,n)
```

Out[123]: array([3, 4, 5])

In [126]: # Set difference

```
np.setdiff1d(m,n)
```

Out[126]: array([1, 2])

In [127]: np.setdiff1d(n,m)

Out[127]: array([6, 7])

In [128]: # set Xor

```
np.setxor1d(m,n)
```

Out[128]: array([1, 2, 6, 7])

In [129]: # in 1D ( Like membership operator)

```
np.in1d(m,1)
```

Out[129]: array([ True, False, False, False, False])

In [131]: m[np.in1d(m,1)]

Out[131]: array([1])

In [130]: np.in1d(m,10)

Out[130]: array([False, False, False, False, False])

## np.clip

numpy.clip() function is used to **Clip (limit) the values** in an array.

In [132]: # code

```
a
```

Out[132]: array([110, 530, 15, 44, 33, 39, 76, 60, 68, 12, 87, 66, 74, 10, 98])

In [133]: np.clip(a, a\_min=15 , a\_max =50)

Out[133]: array([50, 50, 15, 44, 33, 39, 50, 50, 50, 15, 50, 50, 15, 50])

it clips the minimum data to 15 and replaces everything below data to 15 and maximum to 50

## np.swapaxes

numpy.swapaxes() function **interchange two axes** of an array.

```
In [137]: arr = np.array([[1, 2, 3], [4, 5, 6]])
swapped_arr = np.swapaxes(arr, 0, 1)
```

```
In [138]: arr
```

```
Out[138]: array([[1, 2, 3],
                  [4, 5, 6]])
```

```
In [139]: swapped_arr
```

```
Out[139]: array([[1, 4],
                  [2, 5],
                  [3, 6]])
```

```
In [140]: print("Original array:")
print(arr)
```

```
Original array:
[[1 2 3]
 [4 5 6]]
```

```
In [141]: print("Swapped array:")
print(swapped_arr)
```

```
Swapped array:
[[1 4]
 [2 5]
 [3 6]]
```

```
In [ ]:
```

**Download Machine Learning Study Material:**

# Python Library – NumPy

**NumPy** is a general-purpose array-processing Python library which provides handy methods/functions for working n-dimensional arrays. NumPy is a short form for “**Numerical Python**“. It provides various computing tools such as comprehensive mathematical functions, and linear algebra routines.

NumPy developed by **Travis Olliphant** in **2005**.

## Install NumPy Library

```
In [1]: pip install numpy
```

```
Requirement already satisfied: numpy in c:\users\user\anaconda3\lib\site-packages (1.2
```

```
1.5)
```

```
Note: you may need to restart the kernel to use updated packages.
```

## Import NumPy Library

```
In [2]: import numpy as np
```

```
In [3]: a = np.arange(15).reshape(3,5)
```

```
a
```

```
Out[3]: array([[ 0,  1,  2,  3,  4],
 [ 5,  6,  7,  8,  9],
 [10, 11, 12, 13, 14]])
```

```
a.shape (3, 5) a.ndim 2 a.dtype.name 'int64' a.itemsize 8 a.size 15 type(a) <class 'numpy.ndarray'> b =
np.array([6, 7, 8]) b array([6, 7, 8]) type(b) <class 'numpy.ndarray'>
```

```
In [4]: # Shape
```

```
a.shape
```

```
Out[4]: (3, 5)
```

```
In [5]: # dimension
```

```
a.ndim
```

```
Out[5]: 2
```

```
In [6]: # data type
```

```
a.dtype
```

```
Out[6]: dtype('int32')
```

```
In [7]: # item size
```

```
a.size
```

```
Out[7]: 15
```

```
In [8]: # type
```

```
print(type(a))
```

```
<class 'numpy.ndarray'>
```

## Array Creation

```
In [9]: b = np.array([2,4,6,8])
```

```
b
```

```
Out[9]: array([2, 4, 6, 8])
```

```
In [10]: # Shape, item Size, Data Type, Type
```

```
print(f"Shape : {b.shape}")
print(f"Size : {b.size}")
print(f"Data Type : {b.dtype}")
print(f"Type : {type(b)}")
```

```
Shape : (4,)
```

```
Size : 4
```

```
Data Type : int32
```

```
Type : <class 'numpy.ndarray'>
```

```
In [11]: c = np.array([1.3, 2.5, 3.4, 4.6])
```

```
c
```

```
Out[11]: array([1.3, 2.5, 3.4, 4.6])
```

```
In [12]: # Shape, item Size, Data Type, Type
```

```
print(f"Shape : {c.shape}")
print(f"Size : {c.size}")
print(f"Data Type : {c.dtype}")
print(f"Type : {type(c)}")
```

```
Shape : (4,)
```

```
Size : 4
```

```
Data Type : float64
```

```
Type : <class 'numpy.ndarray'>
```

```
In [13]: # 2 dimension array creation  
  
d = np.array([(1,3,5,7),(2,4,6,8)])  
  
d
```

```
Out[13]: array([[1, 3, 5, 7],  
                 [2, 4, 6, 8]])
```

```
In [14]: # Shape, item Size, Data Type, Type  
  
print(f"Shape : {d.shape}")  
print(f"Size : {d.size}")  
print(f"Data Type : {d.dtype}")  
print(f"Type : {type(d)}")  
  
Shape : (2, 4)  
Size : 8  
Data Type : int32  
Type : <class 'numpy.ndarray'>
```

```
In [15]: # numpy array create with data type  
  
e = np.array([10,20,30,40], dtype=complex)  
  
e
```

```
Out[15]: array([10.+0.j, 20.+0.j, 30.+0.j, 40.+0.j])
```

```
In [16]: # data type  
  
e.dtype
```

```
Out[16]: dtype('complex128')
```

## Zeros, Ones, Empty Arrays

```
In [17]: ab = np.zeros((3,4))  
  
ab
```

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

```
In [18]: ac = np.ones((2,4))  
  
ac
```

```
Out[18]: array([[1., 1., 1., 1.],  
                 [1., 1., 1., 1.]])
```

```
In [19]: # data type, shape, size
```

```
print(f"Zeros Array\nData Type : {ab.dtype}")
print(f"Item Size : {ab.size}")
print(f"Shape      : {ab.shape}\n")
print(f"Ones Array\nData Type : {ac.dtype}")
print(f"Item Size : {ac.size}")
print(f"Shape      : {ac.shape}")
```

```
Zeros Array
Data Type : float64
Item Size : 12
Shape      : (3, 4)
```

```
Ones Array
Data Type : float64
Item Size : 8
Shape      : (2, 4)
```

```
In [20]: one = np.ones((3,3), dtype='int8')
```

```
print(f"Ones Array\n\n{one}")
print("\nData Type :", one.dtype)
print("Shape      :", one.shape)
print('Size       :', one.size)
```

```
Ones Array
```

```
[[1 1 1]
 [1 1 1]
 [1 1 1]]
```

```
Data Type : int8
Shape      : (3, 3)
Size       : 9
```

```
In [21]: zero = np.zeros((5,5), dtype='int16')
```

zero[2,2] = 5      **Download Machine Learning Study Material:**

`print(zero)`      <https://t.me/AIMLDeepThaught>

```
[[0 0 0 0 0]
 [0 0 0 0 0]
 [0 0 5 0 0]
 [0 0 0 0 0]
 [0 0 0 0 0]]
```

```
In [22]: three = np.zeros((3,3))

three[1,1] = 5

inone = np.ones((5,5))

inone[1:-1,1:4] = three

print(f"Zeros with 5 \n\n{three}\n")
print(f"Ones & Zeros with 5 \n\n{inone}")
```

Zeros with 5

```
[[0. 0. 0.]
 [0. 5. 0.]
 [0. 0. 0.]]
```

Ones & Zeros with 5

```
[[1. 1. 1. 1. 1.]
 [1. 0. 0. 0. 1.]
 [1. 0. 5. 0. 1.]
 [1. 0. 0. 0. 1.]
 [1. 1. 1. 1. 1.]]
```

```
In [23]: out = np.zeros((5,5))

inner = np.ones((3,3))

inner[1,1] = 0

out[1:4,1:4] = inner

print(out)
```

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

```
In [24]: unequal = inner      Download Machine Learning Study Material:  
unequal[1,1] = 0          https://t.me/AIMLDeepThaught  
print(inner)
```

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

```
In [25]: # copy numpy
```

```
onn = np.ones((3,2), dtype='int8')

onc = onn.copy()

onc[1,:] = 0

print('Original - \n', onn)
print('\nCopy - \n', onc)
```

```
Original -
[[1 1]
 [1 1]
 [1 1]]
```

```
Copy -
[[1 1]
 [0 0]
 [1 1]]
```

```
In [26]: empty = np.empty((2,3)) #Empty gernerates random numbers
```

```
empty
```

```
Out[26]: array([[6.23042070e-307, 4.67296746e-307, 1.69121096e-306],
 [9.34609111e-307, 1.42413555e-306, 1.78019082e-306]])
```

```
In [27]: rang = np.arange(10,50, 10)
```

```
rang
```

```
Out[27]: array([10, 20, 30, 40])
```

```
In [28]: arange = np.arange(5)
```

```
arange
```

```
Out[28]: array([0, 1, 2, 3, 4])
```

```
In [29]: rg = np.random.default_rng(1) # create instance of default random number generator
```

```
d_random = rg.random((2,4))
```

```
d_random
```

```
Out[29]: array([[0.51182162, 0.9504637 , 0.14415961, 0.94864945],
 [0.31183145, 0.42332645, 0.82770259, 0.40919914]])
```

```
In [30]: d_random = rg.random((3,4))
```

```
d_random
```

```
Out[30]: array([[0.54959369, 0.02755911, 0.75351311, 0.53814331],
 [0.32973172, 0.7884287 , 0.30319483, 0.45349789],
 [0.1340417 , 0.40311299, 0.20345524, 0.26231334]])
```

```
In [31]: rng_reshape = np.arange(15).reshape(5,3)
```

```
rng_reshape
```

```
Out[31]: array([[ 0,  1,  2],
   [ 3,  4,  5],
   [ 6,  7,  8],
   [ 9, 10, 11],
   [12, 13, 14]])
```

```
In [32]: abc = np.arange(9).reshape(3,3)
```

```
abc
```

```
Out[32]: array([[0, 1, 2],
   [3, 4, 5],
   [6, 7, 8]])
```

```
In [33]: # Sum of each column
```

```
abc.sum(axis=0)
```

```
Out[33]: array([ 9, 12, 15])
```

```
In [34]: # Sum of each row
```

```
abc.sum(axis=1)
```

```
Out[34]: array([ 3, 12, 21])
```

```
In [35]: # Shape, item Size, Data Type, Type
```

```
print(f"Shape : {rang.shape}")
print(f"Size : {rang.size}")
print(f"Data Type : {rang.dtype}")
print(f"Type : {type(rang)}")
```

```
Shape : (4,)
Size : 4
Data Type : int32
Type : <class 'numpy.ndarray'>
```

## Numpy Zeros Like

```
In [36]: x_like = np.arange(9).reshape(3,3)
```

```
x_like
```

```
Out[36]: array([[0, 1, 2],
   [3, 4, 5],
   [6, 7, 8]])
```

```
In [37]: zeros_like = np.zeros_like(x_like)

zeros_like
```

```
Out[37]: array([[0, 0, 0],
                 [0, 0, 0],
                 [0, 0, 0]])
```

## Numpy Ones Like

```
In [38]: ones_like = np.ones_like(zeros_like)

ones_like
```

```
Out[38]: array([[1, 1, 1],
                 [1, 1, 1],
                 [1, 1, 1]])
```

## Numpy Full Like

```
In [39]: full_like = np.full_like(ones_like, 7) # Full Like with Array Value

full_like
```

```
Out[39]: array([[7, 7, 7],
                 [7, 7, 7],
                 [7, 7, 7]])
```

```
In [40]: ones_like = np.ones_like(zeros_like, order='C')

ones_like
```

```
Out[40]: array([[1, 1, 1],
                 [1, 1, 1],
                 [1, 1, 1]])
```

```
In [41]: from numpy import pi
```

```
In [42]: ln = np.linspace(2,4,5) # 5 numbers from 2 to 4

print(ln)

[2. 2.5 3. 3.5 4.]
```

```
In [43]: li_pi = np.linspace(0, 2 * pi, 12)

fx = np.sin(li_pi)

fx
```

```
Out[43]: array([ 0.0000000e+00,  5.40640817e-01,  9.09631995e-01,  9.89821442e-01,
                 7.55749574e-01,  2.81732557e-01, -2.81732557e-01, -7.55749574e-01,
                 -9.89821442e-01, -9.09631995e-01, -5.40640817e-01, -2.44929360e-16])
```

## Basic Operations

```
In [44]: a = np.array([10,20,30,40])
```

```
b = np.arange(4)
```

Download Machine Learning Study Material:

```
add = a + b
```

<https://t.me/AIMLDeepThought>

```
sub = a - b # Subtraction (a-b)
```

```
mul = a * b
```

```
print(f"Array A : {a}\nArray B : {b}\n")
```

```
print(f"Addition (a+b)\n{add}\n")
```

```
print(f"Subtraction (a-b)\n{sub}\n")
```

```
print(f"Multiplication (a*b)\n{mul}\n")
```

```
Array A : [10 20 30 40]
```

```
Array B : [0 1 2 3]
```

```
Addition (a+b)
```

```
[10 21 32 43]
```

```
Subtraction (a-b)
```

```
[10 19 28 37]
```

```
Multiplication (a*b)
```

```
[ 0 20 60 120]
```

```
In [45]: # a array square
```

```
a_2 = a ** 2
```

```
a_2
```

```
Out[45]: array([ 100, 400, 900, 1600], dtype=int32)
```

```
In [46]: # check values True or False
```

```
a_2 < 500
```

```
Out[46]: array([ True, True, False, False])
```

```
In [47]: a = np.array([[1,1],  
[0,1]])
```

```
b = np.array([[2,0],  
[3,4]])
```

```
a * b # elementwise product
```

```
Out[47]: array([[2, 0],  
[0, 4]])
```

```
In [48]: # matrix product
```

```
a @ b
```

```
Out[48]: array([[5, 4],  
                 [3, 4]])
```

```
In [49]: a.dot(b) # another matrix product
```

```
Out[49]: array([[5, 4],  
                 [3, 4]])
```

```
In [50]: rg = np.random.default_rng(1) # create instance of default random number generator  
a = np.ones((2, 3), dtype=int)  
b = rg.random((2, 3))  
a *= 3  
a
```

```
Out[50]: array([[3, 3, 3],  
                 [3, 3, 3]])
```

```
In [51]: b += a  
b
```

```
Out[51]: array([[3.51182162, 3.9504637 , 3.14415961],  
                 [3.94864945, 3.31183145, 3.42332645]])
```

```
In [52]: a += b # b is not automatically converted to integer type
```

```
a
```

```
-----  
UFuncTypeError                                                 Traceback (most recent call last)  
~\AppData\Local\Temp\ipykernel_10432\1729391664.py in <module>  
----> 1 a += b # b is not automatically converted to integer type  
      2  
      3 a
```

```
UFuncTypeError: Cannot cast ufunc 'add' output from dtype('float64') to dtype('int32')  
with casting rule 'same_kind'
```

```
In [53]: a = np.array([10,15,20,25,30])
```

```
a.sum()
```

```
Out[53]: 100
```

```
In [54]: a.min()
```

```
Out[54]: 10
```

```
In [55]: a.max()
```

```
Out[55]: 30
```

```
In [56]: b = np.arange(12).reshape(3, 4)
```

```
b
```

```
Out[56]: array([[ 0,  1,  2,  3],
                 [ 4,  5,  6,  7],
                 [ 8,  9, 10, 11]])
```

```
In [57]: b.sum(axis=0) # sum of each column
```

```
Out[57]: array([12, 15, 18, 21])
```

```
In [58]: b.sum(axis=1) # sum of each row
```

```
Out[58]: array([ 6, 22, 38])
```

```
In [59]: b.min(axis=0) # min of each column
```

```
Out[59]: array([0, 1, 2, 3])
```

```
In [60]: b.min(axis=1) # min of each row
```

```
Out[60]: array([0, 4, 8])
```

```
In [61]: b.max(axis=0) # max of each column
```

```
Out[61]: array([ 8,  9, 10, 11])
```

```
In [62]: b.max(axis=1) # max of each row
```

```
Out[62]: array([ 3,  7, 11])
```

```
In [63]: b.cumsum(axis=0) # cumulative sum along each column
```

```
Out[63]: array([[ 0,  1,  2,  3],
                 [ 4,  6,  8, 10],
                 [12, 15, 18, 21]], dtype=int32)
```

```
In [64]: b.cumsum(axis=1) # cumulative sum along each row
```

```
Out[64]: array([[ 0,  1,  3,  6],
                 [ 4,  9, 15, 22],
                 [ 8, 17, 27, 38]], dtype=int32)
```

## Universal Function

```
In [65]: a = np.array([10,20,30])
```

```
np.exp(a)
```

```
Out[65]: array([2.20264658e+04, 4.85165195e+08, 1.06864746e+13])
```

```
In [66]: np.sqrt(81)
```

```
Out[66]: 9.0
```

```
In [67]: b = np.array([4,9,16,25])  
x = np.sqrt(b)  
x
```

```
Out[67]: array([2., 3., 4., 5.])
```

```
In [68]: y = np.array([1.,-2.,3.,-3.])  
np.add(x, y)
```

```
Out[68]: array([3., 1., 7., 2.])
```

## Array Indexing / Slicing

```
In [69]: a = np.array([2,4,6,8]) # 1D array  
  
print(a)  
print(a.shape)
```

```
[2 4 6 8]  
(4,)
```

```
In [70]: a[:2] # print first two elements
```

```
Out[70]: array([2, 4])
```

```
In [71]: a[-1] # print last one elements
```

```
Out[71]: 8
```

```
In [72]: b = np.array([[1,2,3,4],[5,6,7,8]]) # 2D array  
  
print(b)  
print(b.shape)
```

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

```
In [73]: b[0][:2] # print first two elements of first row
```

```
Out[73]: array([1, 2])
```

```
In [74]: b[1, :2] # print first two elements of second row
```

```
Out[74]: array([5, 6])
```

```
In [75]: b[1, 2] # print 7
```

```
Out[75]: 7
```

```
In [76]: c = np.array([[1,2,3],[4,5,6],[[7,8,9],[10,11,12]]]) # 3D array
```

```
print(c)
print(c.shape)
```

```
[[[ 1  2  3]
 [ 4  5  6]]
```

```
[[ 7  8  9]
 [10 11 12]]]
```

```
(2, 2, 3)
```

```
In [77]: c[0]
```

```
Out[77]: array([[1, 2, 3],
 [4, 5, 6]])
```

```
In [78]: c[1]
```

```
Out[78]: array([[ 7,  8,  9],
 [10, 11, 12]])
```

```
In [79]: c[0,0]
```

```
Out[79]: array([1, 2, 3])
```

```
In [80]: c[1,0]
```

```
Out[80]: array([7, 8, 9])
```

```
In [81]: c[0, 1, -1]
```

```
Out[81]: 6
```

```
In [82]: c[:, 1, 0]
```

```
Out[82]: array([ 4, 10])
```

```
In [83]: c[1, :, -1]
```

```
Out[83]: array([ 9, 12])
```

## NumPy Data Types

```
In [84]: a = np.array(['Data Science', 'Machine Learning', 'Deep Learning', 'AI' ])
```

```
print(a)
print(a.dtype)
```

```
['Data Science' 'Machine Learning' 'Deep Learning' 'AI']
```

```
<U16
```

```
In [85]: a.dtype
```

```
Out[85]: dtype('<U16')
```

```
In [86]: b = np.array([100,200,300])
```

```
b.dtype
```

```
Out[86]: dtype('int32')
```

```
In [87]: c = np.array([2.05, 3.0, 4.01, 5.6])
```

```
c.dtype
```

```
Out[87]: dtype('float64')
```

```
In [88]: d = np.array([True, True, False, True])
```

```
d.dtype
```

```
Out[88]: dtype('bool')
```

```
In [89]: e = np.array([1.0 + 2.0j, 1.5 + 2.5j])
```

```
e.dtype
```

```
Out[89]: dtype('complex128')
```

## Numpy Copy / View

The copy owns the data and any changes made to the copy will not affect original array, and any changes made to the original array will not affect the copy.

The view does not own the data and any changes made to the view will affect the original array, and any changes made to the original array will affect the view.

```
In [90]: a = np.array([1,2,3,4,5])
```

```
x = a.copy()
```

```
a[1] = 10
```

```
print(a)
print(x)
```

```
[ 1 10  3  4  5]
[1 2 3 4 5]
```

```
In [91]: a = np.array([1,2,3,4,5])
x = a.copy()
x[1] = 7
print(a)
print(x)
```

```
[1 2 3 4 5]
[1 7 3 4 5]
```

```
In [92]: a = np.array([1,2,3,4,5])
x = a.view()
a[1] = 10
print(a)
print(x)
```

```
[ 1 10  3  4  5]
[ 1 10  3  4  5]
```

```
In [93]: a = np.array([1,2,3,4,5])
x = a.view()
x[-1] = 20
print(a)
print(x)
```

```
[ 1  2  3  4 20]
[ 1  2  3  4 20]
```

## Reshape NumPy Array

```
In [94]: a = np.arange(12)
a
```

```
Out[94]: array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])
```

```
In [95]: a2 = a.reshape(4,3) # Reshape 1D array to 2D array
a2
```

```
Out[95]: array([[ 0,  1,  2],
 [ 3,  4,  5],
 [ 6,  7,  8],
 [ 9, 10, 11]])
```

```
In [96]: a3 = a.reshape(2,3,2) # Reshape 1D to 3D array  
a3
```

```
Out[96]: array([[[ 0,  1],  
                  [ 2,  3],  
                  [ 4,  5]],  
  
                 [[ 6,  7],  
                  [ 8,  9],  
                  [10, 11]])
```

```
In [97]: print(f"a Shape: {a.shape}")  
print(f"a2 Shape: {a2.shape}")  
print(f"a3 Shape: {a3.shape}")  
  
a Shape: (12,)  
a2 Shape: (4, 3)  
a3 Shape: (2, 3, 2)
```

## NumPy Array Iterating

```
In [98]: for i in a:  
    print(i)  
  
0  
1  
2  
3  
4  
5  
6  
7  
8  
9  
10  
11
```

```
In [99]: for x in a2:  
    print(x)  
  
[0 1 2]  
[3 4 5]  
[6 7 8]  
[ 9 10 11]
```

```
In [100]: for y in a3:  
    print(y)  
  
[[0 1]  
 [2 3]  
 [4 5]]  
[[ 6  7]  
 [ 8  9]  
 [10 11]]
```

## NumPy Array Join

```
In [101]: a = np.array([1,2,3])
b = np.array([4,5,6])
ab = np.concatenate((a, b))
ab
```

```
Out[101]: array([1, 2, 3, 4, 5, 6])
```

```
In [102]: a = np.array([[1,2],[3,4]])
b = np.array([[5,6],[7,8]])
ab = np.concatenate((a, b))
ab
```

```
Out[102]: array([[1, 2],
 [3, 4],
 [5, 6],
 [7, 8]])
```

```
In [103]: a = np.array([[1,2],[3,4]])
b = np.array([[5,6],[7,8]])
ab = np.concatenate((a, b), axis=1)
ab
```

```
Out[103]: array([[1, 2, 5, 6],
 [3, 4, 7, 8]])
```

```
In [104]: # Stack Function
a = np.array([1, 2, 3])
b = np.array([4, 5, 6])
ab = np.stack((a,b), axis=1)
ab
```

```
Out[104]: array([[1, 4],
 [2, 5],
 [3, 6]])
```

```
In [105]: a = np.array([1, 2, 3])
b = np.array([4, 5, 6])
ab = np.hstack((a,b))
ab
```

```
Out[105]: array([1, 2, 3, 4, 5, 6])
```

```
In [106]: a = np.array([[1,2],[3,4]])
b = np.array([[5,6],[7,8]])
ab = np.hstack((a, b))
ab
```

```
Out[106]: array([[1, 2, 5, 6],
 [3, 4, 7, 8]])
```

```
In [107]: a = np.array([1, 2, 3])
b = np.array([4, 5, 6])
ab = np.vstack((a,b))
ab
```

```
Out[107]: array([[1, 2, 3],
 [4, 5, 6]])
```

```
In [108]: a = np.array([1, 2, 3])
b = np.array([4, 5, 6])
ab = np.dstack((a,b))
ab
```

```
Out[108]: array([[[1, 4],
 [2, 5],
 [3, 6]]])
```

## NumPy Array Split

```
In [109]: a = np.array([1, 2, 3, 4, 5, 6])
ab = np.array_split(a, 2)
ab
```

```
Out[109]: [array([1, 2, 3]), array([4, 5, 6])]
```

```
In [110]: a = np.array([1, 2, 3, 4, 5, 6])
ab = np.array_split(a, 3)
ab
```

```
Out[110]: [array([1, 2]), array([3, 4]), array([5, 6])]
```

```
In [111]: a = np.array([1, 2, 3, 4, 5, 6])
ab = np.array_split(a, 4)
ab
```

```
Out[111]: [array([1, 2]), array([3, 4]), array([5]), array([6])]
```

## Split Into Arrays

```
In [112]: ab[0]
```

```
Out[112]: array([1, 2])
```

```
In [113]: ab[1]
```

```
Out[113]: array([3, 4])
```

```
In [114]: ab[2]
```

```
Out[114]: array([5])
```

```
In [115]: ab[3]
```

```
Out[115]: array([6])
```

## 2D Arrays Split

```
In [116]: a = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12], [13, 14, 15], [16, 17, 18]])
ab = np.array_split(a, 2)
ab
```

```
Out[116]: [array([[1, 2, 3],
                  [4, 5, 6],
                  [7, 8, 9]]),
           array([[10, 11, 12],
                  [13, 14, 15],
                  [16, 17, 18]])]
```

```
In [117]: ab[0]
```

```
Out[117]: array([[1, 2, 3],
                  [4, 5, 6],
                  [7, 8, 9]])
```

```
In [118]: ab[1]
```

```
Out[118]: array([[10, 11, 12],  
                 [13, 14, 15],  
                 [16, 17, 18]])
```

```
In [119]: a = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12], [13, 14, 15], [16, 17, 18]])  
  
ab = np.array_split(a, 3)  
  
ab
```

```
Out[119]: [array([[1, 2, 3],  
                  [4, 5, 6]]),  
           array([[ 7,  8,  9],  
                  [10, 11, 12]]),  
           array([[13, 14, 15],  
                  [16, 17, 18]])]
```

```
In [120]: ab[0]
```

```
Out[120]: array([[1, 2, 3],  
                  [4, 5, 6]])
```

```
In [121]: ab[1]
```

```
Out[121]: array([[ 7,  8,  9],  
                  [10, 11, 12]])
```

```
In [122]: ab[2]
```

```
Out[122]: array([[13, 14, 15],  
                  [16, 17, 18]])
```

```
In [123]: a = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]])  
  
ab = np.array_split(a, 4)  
  
ab
```

```
Out[123]: [array([[1, 2, 3]]),  
           array([[4, 5, 6]]),  
           array([[7, 8, 9]]),  
           array([[10, 11, 12]])]
```

```
In [124]: ab[0]
```

```
Out[124]: array([[1, 2, 3]])
```

```
In [125]: ab[1]
```

```
Out[125]: array([[4, 5, 6]])
```

```
In [126]: ab[2]
```

```
Out[126]: array([[7, 8, 9]])
```

```
In [127]: a = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12], [13, 14, 15], [16, 17, 18]])
ab = np.hsplit(a, 3)
ab
```

```
Out[127]: [array([[ 1],
                   [ 4],
                   [ 7],
                   [10],
                   [13],
                   [16]]),
            array([[ 2],
                   [ 5],
                   [ 8],
                   [11],
                   [14],
                   [17]]),
            array([[ 3],
                   [ 6],
                   [ 9],
                   [12],
                   [15],
                   [18]])]
```

```
In [128]: a = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
ab = np.hsplit(a, 3)
ab
```

```
Out[128]: [array([[1],
                   [4],
                   [7]]),
            array([[2],
                   [5],
                   [8]]),
            array([[3],
                   [6],
                   [9]])]
```

```
In [129]: a = np.array([[1,2],[3,4],[5,6]])
ab = np.hsplit(a, 2)
ab
```

```
Out[129]: [array([[1],
                   [3],
                   [5]]),
            array([[2],
                   [4],
                   [6]])]
```

**Download Machine Learning Study Material:**

<https://t.me/AIMLDeepThaught>

```
In [130]: a = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9])
ab = np.where(a == 5)
ab
```

```
Out[130]: (array([4], dtype=int64),)
```

```
In [131]: a = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9])
ab = np.where(a == 8)
ab
```

```
Out[131]: (array([7], dtype=int64),)
```

```
In [132]: a = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9])
ab = np.where(a%2 == 0)
ab
```

```
Out[132]: (array([1, 3, 5, 7], dtype=int64),)
```

```
In [133]: a = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13])
ab = np.where(a%2 == 1)
ab
```

```
Out[133]: (array([ 0,  2,  4,  6,  8, 10, 12], dtype=int64),)
```

```
In [134]: a = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13])
ab = np.searchsorted(a, 3)
ab
```

```
Out[134]: 2
```

```
In [135]: a = np.array([6, 7, 8, 9, 10, 11, 12, 13])
ab = np.searchsorted(a, 12, side='right')
ab
```

```
Out[135]: 7
```

```
In [136]: a = np.array([6, 7, 8, 9, 10, 11, 12, 13])
ab = np.searchsorted(a, 12)
ab
```

```
Out[136]: 6
```

```
In [137]: a = np.array([6, 7, 8, 9, 10, 11, 12, 13])
          ab = np.searchsorted(a, [1,3,8,9,10])
          ab
```

```
Out[137]: array([0, 0, 2, 3, 4], dtype=int64)
```

## NumPy Array Sort

```
In [138]: a = np.array([14, 1, 11, 3, 6, 4, 7, 12, 13, 8, 15, 9, 10, 0, 2, 5])
          np.sort(a)
```

```
Out[138]: array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15])
```

```
In [139]: a = np.array(['Cat', 'Zoo', 'Football', 'Dog', 'Eye', 'Apple'])
          np.sort(a)
```

```
Out[139]: array(['Apple', 'Cat', 'Dog', 'Eye', 'Football', 'Zoo'], dtype='|<U8')
```

```
In [140]: a = np.array([False, True, False, False, True, False])
          np.sort(a)
```

```
Out[140]: array([False, False, False, False, True, True])
```

```
In [141]: a2 = np.array([[3, 2, 4], [5, 0, 1]])
          np.sort(a2)
```

```
Out[141]: array([[2, 3, 4],
                  [0, 1, 5]])
```

```
In [142]: arr = np.array([41, 42, 43, 44])

          # Create an empty list
          filter_arr = []

          # go through each element in arr
          for element in arr:
              # if the element is higher than 42, set the value to True, otherwise False:
              if element > 42:
                  filter_arr.append(True)
              else:
                  filter_arr.append(False)

          newarr = arr[filter_arr]

          print(filter_arr)
          print(newarr)

[False, False, True, True]
[43 44]
```

```
In [143]: arr = np.array([1, 2, 3, 4, 5, 6, 7])
```

```
# Create an empty list
filter_arr = []

# go through each element in arr
for element in arr:
    # if the element is completely divisible by 2, set the value to True, otherwise False
    if element % 2 == 0:
        filter_arr.append(True)
    else:
        filter_arr.append(False)

newarr = arr[filter_arr]

print(filter_arr)
print(newarr)
```

```
[False, True, False, True, False, True, False]
[2 4 6]
```

```
In [144]: arr = np.array([1, 2, 3, 4, 5, 6, 7])
```

```
# Create an empty list
filter_arr = []

# go through each element in arr
for element in arr:
    # if the element is not completely divisible by 2, set the value to True, otherwise False
    if element % 2 == 1:
        filter_arr.append(True)
    else:
        filter_arr.append(False)

newarr = arr[filter_arr]

print(filter_arr)
print(newarr)
```

```
[True, False, True, False, True, False, True]
[1 3 5 7]
```

```
In [145]: ax = np.array([10, 20, 30, 40, 50])
```

```
filter_arr = ax > 20

new_ax = ax[filter_arr]

print(filter_arr)
print(new_ax)
```

```
[False False  True  True  True]
[30 40 50]
```

## Random Numbers in NumPy

```
In [146]: # import Library
```

```
from numpy import random
```

```
In [147]: x = random.randint(100)
```

```
print(x)
```

```
39
```

```
In [148]: x = random.rand()
```

```
x
```

```
Out[148]: 0.33150184362238155
```

```
In [149]: # Generate a 1-D array containing 5 random integers from 0 to 100:
```

```
x = random.randint(100, size=(3))
```

```
print(x)
```

```
[92 25 61]
```

```
In [150]: # Generate a 2-D array with 2 rows, each row containing 3 random integers from 0 to 100:
```

```
x = random.randint(100, size=(2,3))
```

```
x
```

```
Out[150]: array([[43, 87, 7],  
 [77, 62, 90]])
```

```
In [151]: # Generate a 1-D array containing 5 random floats:
```

```
x = random.rand(5)
```

```
print(x)
```

```
[0.2283375 0.79971818 0.32832239 0.54718548 0.0303609 ]
```

```
In [152]: # Generate a 2-D array with 3 rows, each row containing 2 random numbers:
```

```
x = random.rand(3, 2)
```

```
print(x)
```

```
[[0.94764137 0.43819239]  
 [0.52815542 0.01067087]  
 [0.72109457 0.10579046]]
```

```
In [153]: # Return one of the values in an array:
```

```
x = random.choice([3, 5, 7, 9])  
print(x)
```

```
7
```

```
In [154]: # The choice() method also allows you to return an array of values.
```

```
# Add a size parameter to specify the shape of the array.
```

```
x = random.choice([3, 5, 7, 9], size=(3, 5))  
print(x)
```

```
[[9 7 5 7 3]  
 [5 5 7 7 5]  
 [3 3 5 3 5]]
```

## Random Data Distribution

### What is Data Distribution?

Data Distribution is a list of all possible values, and how often each value occurs. Such lists are important when working with statistics and data science. The random module offer methods that returns randomly generated data distributions.

**Example**Get your own Python Server Generate a 1-D array containing 100 values, where each value has to be 3, 5, 7 or 9.

The probability for the value to be 3 is set to be 0.1

The probability for the value to be 5 is set to be 0.3

The probability for the value to be 7 is set to be 0.6

The probability for the value to be 9 is set to be 0

```
In [155]: x = random.choice([3, 5, 7, 9], p=[0.1, 0.3, 0.6, 0.0], size=(100))
```

```
print(x)
```

```
[5 7 3 7 7 7 3 7 7 7 7 7 5 7 3 7 7 5 7 5 5 7 3 7 7 5 7 7 7 7 7 5 7 5 7 5 7 7  
 7 7 7 7 7 7 7 7 7 5 5 5 5 5 7 7 7 7 7 5 7 3 3 7 5 5 7 7 7 7 5 5 5 5 5 5 5  
 7 7 5 7 7 5 5 7 5 7 7 7 7 7 7 7 5 7 3 5 5 7]
```

```
In [156]: # Same example as above, but return a 2-D array with 3 rows, each containing 5 values.
```

```
x = random.choice([3, 5, 7, 9], p=[0.1, 0.3, 0.6, 0.0], size=(3, 5))
```

```
print(x)
```

```
[[7 7 5 5 3]  
 [5 5 7 7 3]  
 [7 5 7 7 3]]
```

## Random Permutations of Elements

### Shuffling Arrays

```
In [157]: arr = np.array([1, 2, 3, 4, 5])  
random.shuffle(arr) # The shuffle() method makes changes to the original array.  
print(arr)  
[1 2 4 3 5]
```

```
In [158]: # Generating Permutation of Arrays  
arr = np.array([1, 2, 3, 4, 5])  
print(random.permutation(arr)) # The permutation() method returns a re-arranged array (array)  
[1 5 4 2 3]
```

### Normal (Gaussian) Distribution

```
In [159]: x = random.normal(size=(2, 3))  
print(x)  
[[ -1.40144554 -1.35534505  0.4355295 ]  
 [ 1.34332605  0.70828922 -0.80585434]]
```

```
In [160]: # Generate a random normal distribution of size 2x3 with mean at 3 and standard deviation 2  
x = random.normal(loc=3, scale=2, size=(2, 3))  
print(x)  
[[ 2.82806795  3.83998103  3.76165957]  
 [ 0.93354518  3.08895529  1.73914323]]
```

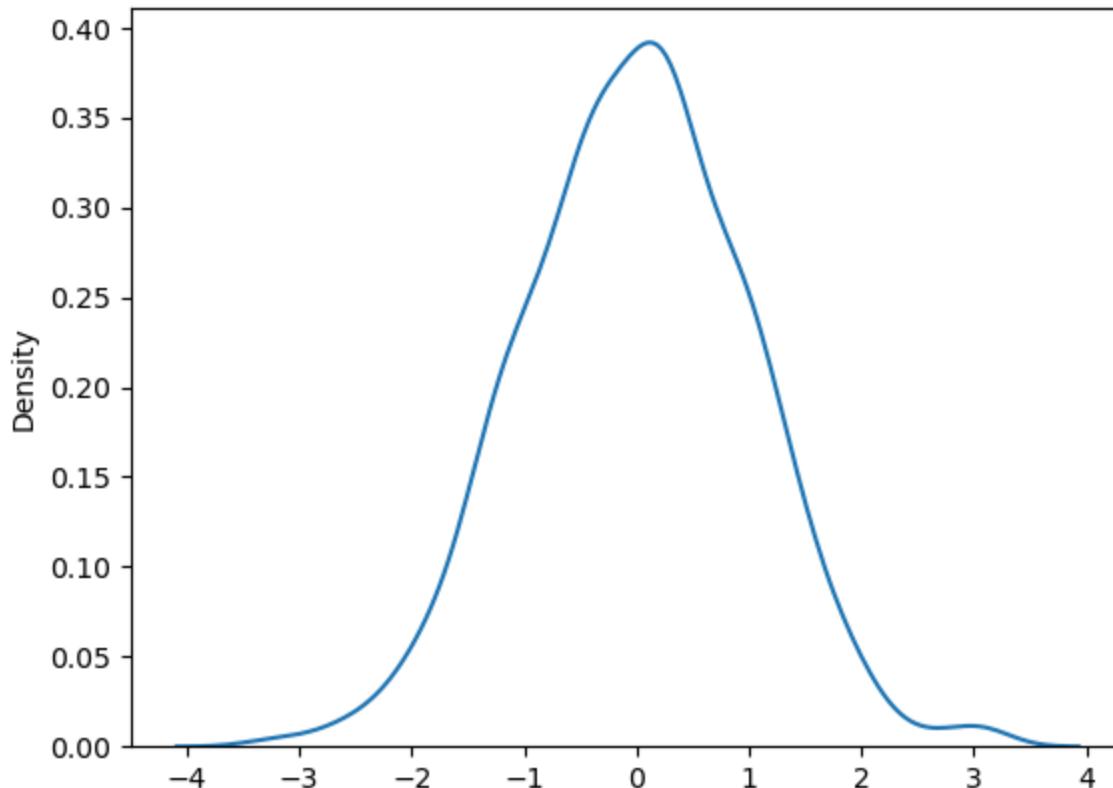
### Visualization of Normal Distribution

```
In [161]: import matplotlib.pyplot as plt
import seaborn as sns

sns.distplot(random.normal(size=1000), hist=False)

plt.show()
```

C:\Users\User\anaconda3\lib\site-packages\seaborn\distributions.py:2619: FutureWarning: `distplot` is a deprecated function and will be removed in a future version. Please adapt your code to use either `displot` (a figure-level function with similar flexibility) or `kdeplot` (an axes-level function for kernel density plots).  
warnings.warn(msg, FutureWarning)



## Binomial Distribution

```
In [162]: x = random.binomial(n=20, p=0.5, size=10)

print(x)
```

[ 8 11 7 9 11 11 11 14 7 7]

## Chi Square Distribution

```
In [163]: x = random.chisquare(df=2, size=(2, 3))

print(x)
```

[[0.74806038 1.03024374 2.75760522]
 [0.33185553 0.10245362 0.27963136]]

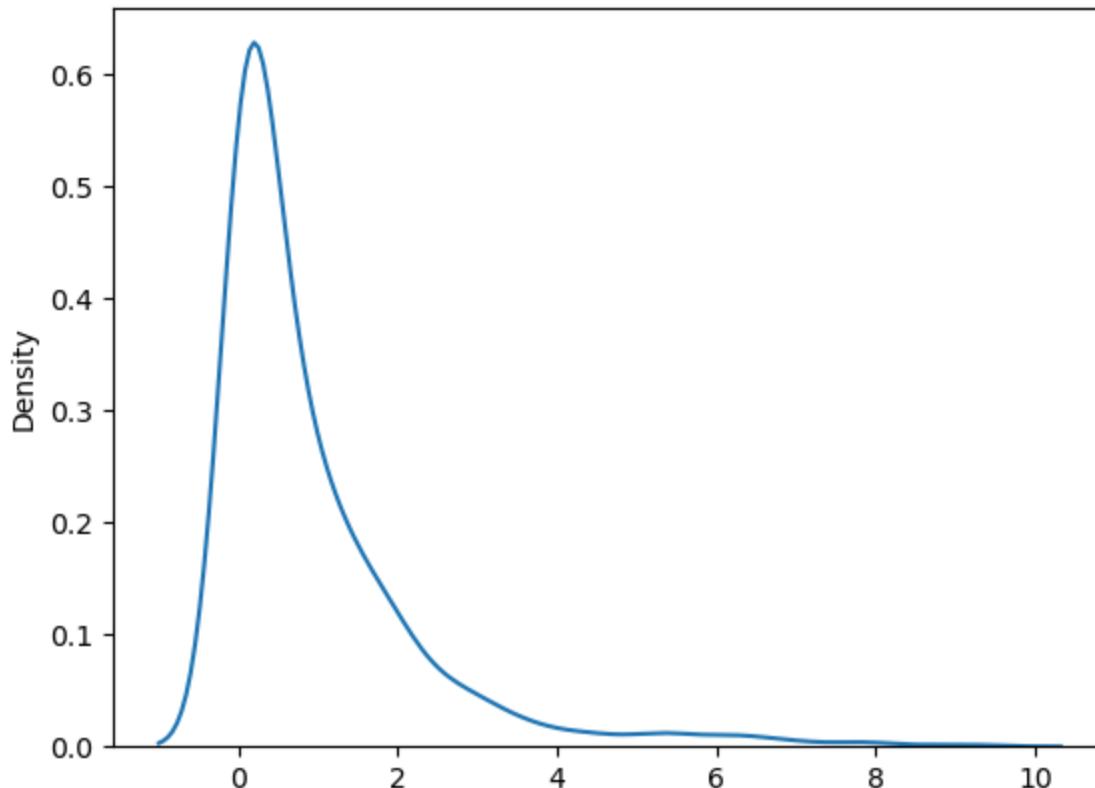
## Visualization of Chi-Square Distribution

```
In [164]: from numpy import random
import matplotlib.pyplot as plt
import seaborn as sns

sns.distplot(random.chisquare(df=1, size=1000), hist=False)

plt.show()
```

C:\Users\User\anaconda3\lib\site-packages\seaborn\distributions.py:2619: FutureWarning: `distplot` is a deprecated function and will be removed in a future version. Please adapt your code to use either `displot` (a figure-level function with similar flexibility) or `kdeplot` (an axes-level function for kernel density plots).  
warnings.warn(msg, FutureWarning)



## Visualization of Poisson Distribution

**Poisson Distribution** has two parameters:

1. lam - rate or known number of occurrences e.g. 2 for above problem.
2. size - The shape of the returned array.

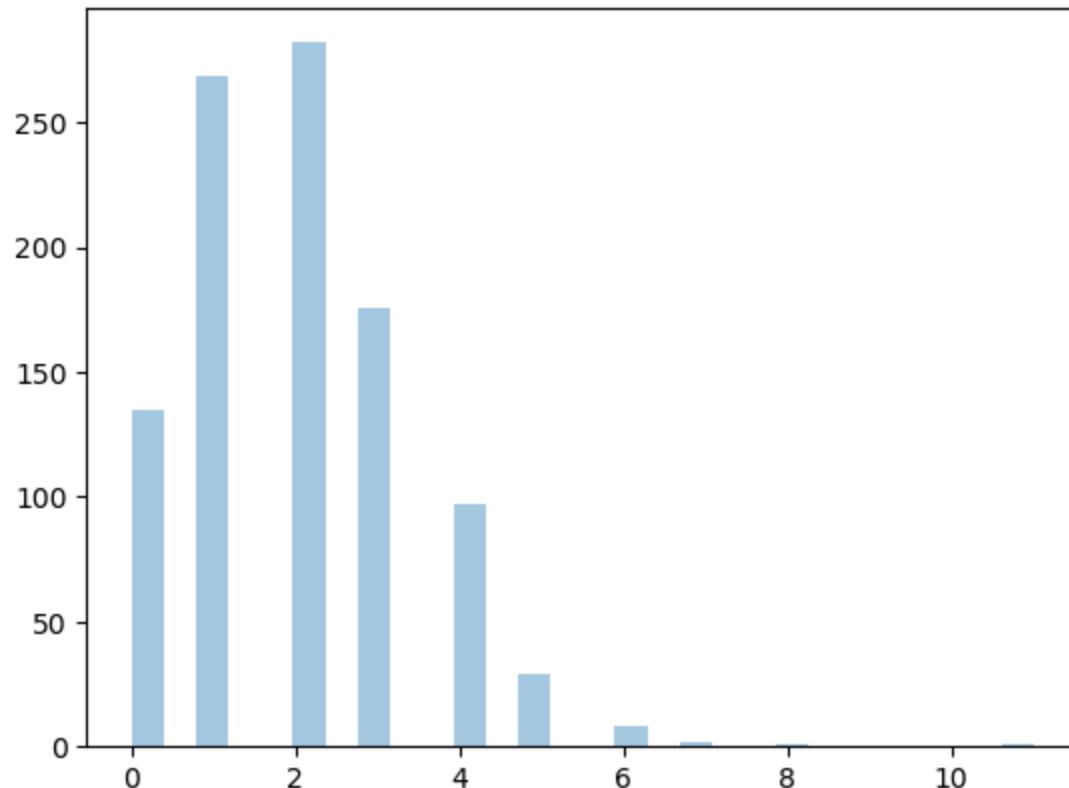
```
In [165]: x = random.poisson(lam=2, size=10)

print(x)

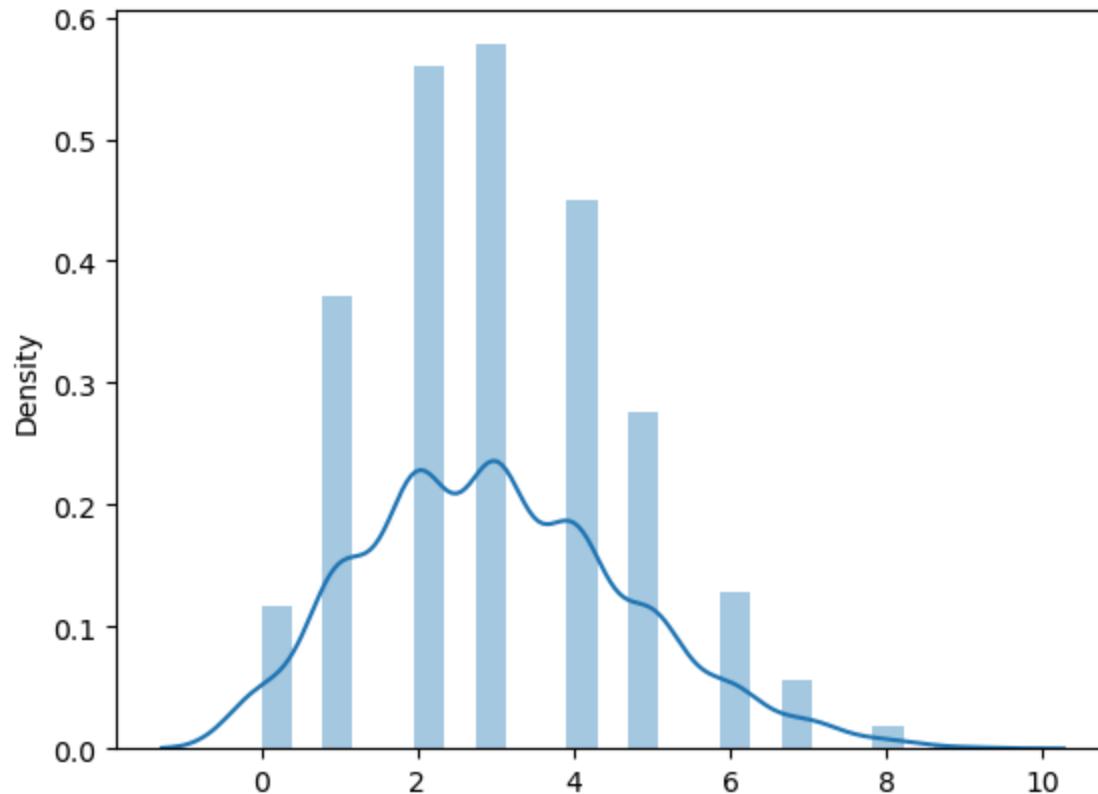
[0 2 1 1 1 1 2 0 3 2]
```

```
In [166]: from numpy import random  
import matplotlib.pyplot as plt  
import seaborn as sns  
  
sns.distplot(random.poisson(lam=2, size=1000), kde=False)  
plt.show()
```

C:\Users\User\anaconda3\lib\site-packages\seaborn\distributions.py:2619: FutureWarning:  
`distplot` is a deprecated function and will be removed in a future version. Please adapt your code to use either `displot` (a figure-level function with similar flexibility) or `histplot` (an axes-level function for histograms).  
warnings.warn(msg, FutureWarning)



```
In [167]: from numpy import random  
import matplotlib.pyplot as plt  
import seaborn as sns  
  
sns.distplot(random.poisson(lam=3, size=1000))  
plt.show()
```



## Uniform Distribution

**Uniform Distribution** has three parameters:

1. a - lower bound - default 0 .0.
2. b - upper bound - default 1.0.
3. size - The shape of the returned array.

```
In [168]: x = random.uniform(size=(2, 3))  
  
print(x)  
  
[[0.82751804 0.00422504 0.97971968]  
[0.12829234 0.24044758 0.90011081]]
```

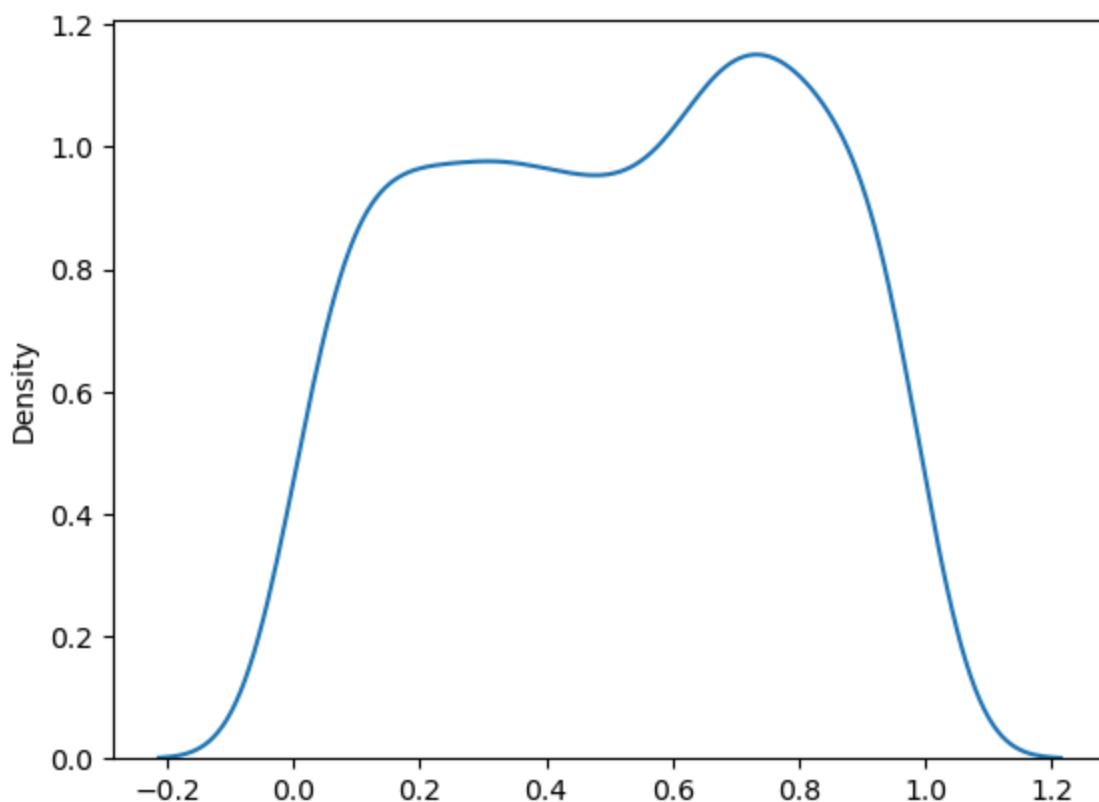
## Visualization of Uniform Distribution

```
In [169]: from numpy import random
import matplotlib.pyplot as plt
import seaborn as sns

sns.distplot(random.uniform(size=1000), hist=False)

plt.show()
```

C:\Users\User\anaconda3\lib\site-packages\seaborn\distributions.py:2619: FutureWarning: `distplot` is a deprecated function and will be removed in a future version. Please adapt your code to use either `displot` (a figure-level function with similar flexibility) or `kdeplot` (an axes-level function for kernel density plots).  
warnings.warn(msg, FutureWarning)

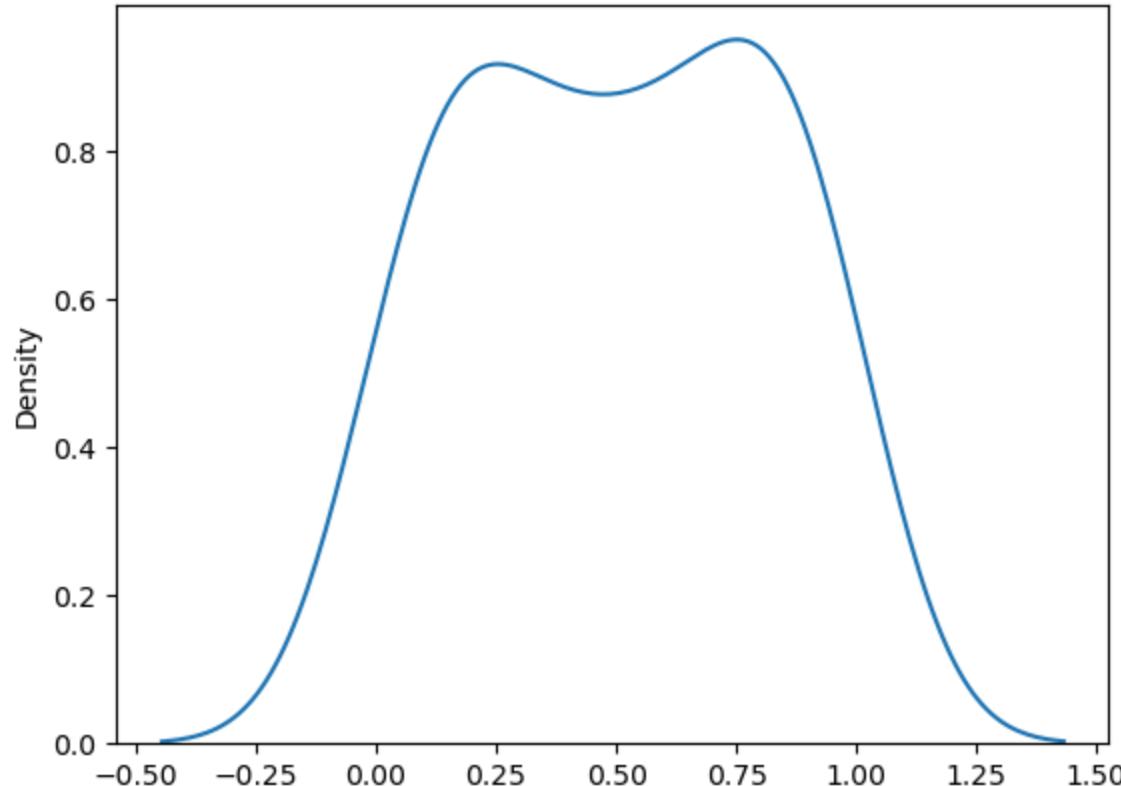


**Download Machine Learning Study Material:**

<https://t.me/AIMLDeepThaught>

```
In [170]: sns.distplot(random.uniform(size=30), hist=False)
plt.show()
```

C:\Users\User\anaconda3\lib\site-packages\seaborn\distributions.py:2619: FutureWarning: `distplot` is a deprecated function and will be removed in a future version. Please adapt your code to use either `displot` (a figure-level function with similar flexibility) or `kdeplot` (an axes-level function for kernel density plots).  
warnings.warn(msg, FutureWarning)



## NumPy ufuncs

```
In [171]: x = [1, 2, 3, 4]
y = [4, 5, 6, 7]
z = []

for i, j in zip(x, y):
    z.append(i + j)
print(z)
```

```
[5, 7, 9, 11]
```

```
In [172]: x = [1, 2, 3, 4]
y = [4, 5, 6, 7]

z = np.add(x, y) # add() function

print(z)
```

```
[ 5  7  9 11]
```

The `frompyfunc()` method takes the following arguments:

1. function - the name of the function.
2. inputs - the number of input arguments (arrays).
3. outputs - the number of output arrays.

```
In [173]: def myadd(x, y):
    return x+y

myadd = np.frompyfunc(myadd, 2, 1)

print(myadd([1, 2, 3, 4], [5, 6, 7, 8]))

[6 8 10 12]
```

```
In [174]: myadd([1, 2, 3, 4], [10,20,30,40])
```

```
Out[174]: array([11, 22, 33, 44], dtype=object)
```

## Check if a Function is a ufunc

```
In [175]: print(type(np.add))

<class 'numpy.ufunc'>
```

```
In [176]: print(type(np.concatenate))

<class 'function'>
```

```
In [177]: if type(np.add) == np.ufunc:
    print('add is ufunc')
else:
    print('add is not ufunc')

add is ufunc
```

```
In [178]: try:
    if type(np.blahblah) == np.ufunc:
        print('blahblah is ufunc')
    else:
        print('blahblah is not ufunc')
except:
    print('blahblah is not ufunc in NumPy.')

blahblah is not ufunc in NumPy.
```

## NumPy Arithmetic

```
In [179]: a = np.array([10,20,30,40])  
b = np.array([100,200,300,400])  
add = np.add(a, b)  
print(add)  
  
[110 220 330 440]
```

```
In [180]: a = np.array([10,20,30,40])  
b = np.array([100,200,300,400])  
subtract = np.subtract(b, a)  
print(subtract)  
  
[ 90 180 270 360]
```

```
In [181]: a = np.array([10,20,30,40])  
b = np.array([100,200,300,400])  
multiply = np.multiply(a, b)  
print(multiply)  
  
[ 1000 4000 9000 16000]
```

```
In [182]: a = np.array([10,20,30,40])  
b = np.array([100,200,300,400])  
divide = np.divide(b, a)  
print(divide)  
  
[10. 10. 10. 10.]
```

```
In [183]: a = np.array([10,20,30,40])  
b = np.array([1,2,3,4])  
power = np.power(a, b)  
print(power)  
  
[ 10 400 27000 2560000]
```

```
In [184]: a = np.array([10,20,30,40])  
b = np.array([3,4,6,7])  
mod = np.mod(a, b)  
print(mod)
```

```
[1 0 0 5]
```

```
In [185]: a = np.array([10,20,30,40])  
b = np.array([3,4,6,7])  
remainder = np.remainder(a, b)  
print(remainder)
```

```
[1 0 0 5]
```

## NumPy Rounding Decimals

```
In [186]: ab = np.trunc([-3.1666, 3.6667])  
print(ab)
```

```
[-3. 3.]
```

```
In [187]: ab = np.fix([-3.1666, 3.6667])  
print(ab)
```

```
[-3. 3.]
```

```
In [188]: ab = np.around([-3.1666, 3.6667])  
print(ab)
```

```
[-3. 4.]
```

```
In [189]: ab = np.floor([-3.1666, 3.6667])  
print(ab)
```

```
[-4. 3.]
```

```
In [190]: ab = np.ceil([-3.1666, 3.6667])  
print(ab)
```

```
[-3. 4.]
```

# NumPy Logs

```
In [191]: import numpy as np  
  
ab = np.arange(1, 6)  
  
log2 = np.log2(ab)  
  
print(log2)  
  
[0.          1.          1.5849625  2.          2.32192809]
```

```
In [192]: ab = np.arange(1, 6)  
  
log10 = np.log10(ab)  
  
print(log10)  
  
[0.          0.30103   0.47712125  0.60205999  0.69897   ]
```

## Natural Log, or Log at Base e

```
In [193]: ab = np.arange(1, 6)  
  
log = np.log(ab)  
  
print(log)  
  
[0.          0.69314718  1.09861229  1.38629436  1.60943791]
```

## Log at Any Base

NumPy does not provide any function to take log at any base, so we can use the ***frompyfunc()*** function along with inbuilt function ***math.log()*** with two input parameters and one output parameter

```
In [194]: from math import log  
import numpy as np  
  
nplog = np.frompyfunc(log, 2, 1)  
  
print(nplog(100, 15))  
  
1.7005483074552052
```

## NumPy Products

```
In [195]: a = np.array([1, 2, 3, 4])  
x = np.prod(a) # 1*2*3*4 = 24  
print(x)
```

```
24
```

```
In [196]: a = np.array([1, 2, 3, 4])  
b = np.array([5, 6, 7, 8])  
x = np.prod([a, b])  
print(x)
```

```
40320
```

```
In [197]: a = np.array([1, 2, 3, 4])  
b = np.array([5, 6, 7, 8])  
x = np.prod([a, b], axis=1) # each row  
print(x)
```

```
[ 24 1680]
```

```
In [198]: a = np.array([1, 2, 3, 4])  
b = np.array([5, 6, 7, 8])  
x = np.prod([a, b], axis=0) # each column  
print(x)
```

```
[ 5 12 21 32]
```

```
In [199]: a = np.array([2, 3, 4, 5])  
x = np.cumprod(a)  
  
print(x)
```

```
[ 2  6 24 120]
```

```
In [200]: a = np.array([1, 2, 3, 4])
b = np.array([5, 6, 7, 8])
x = np.cumprod([a , b])

print(x)
```

```
[ 1  2   6  24 120 720 5040 40320]
```

```
In [201]: a = np.array([1, 2, 3, 4])
b = np.array([5, 6, 7, 8])
x = np.cumprod([a , b], axis=1)

print(x)
```

```
[[ 1   2   6  24]
 [ 5   30  210 1680]]
```

```
In [202]: a = np.array([1, 2, 3, 4])
b = np.array([5, 6, 7, 8])
c = np.array([9, 10, 11, 12])
x = np.cumprod([a , b, c], axis=0)

print(x)
```

```
[[ 1   2   3   4]
 [ 5   12  21  32]
 [ 45  120 231 384]]
```

## NumPy Differences

E.g. for [1, 2, 3, 4], the discrete difference would be [2-1, 3-2, 4-3] = [1, 1, 1]

```
In [203]: a = np.array([10, 15, 25, 5])
diff = np.diff(a)

print(diff)
```

```
[ 5  10 -20]
```

We can perform this operation repeatedly by giving parameter **n**.

E.g. for [1, 2, 3, 4], the discrete difference with n = 2 would be [2-1, 3-2, 4-3] = [1, 1, 1] , then, since n=2, we will do it once more, with the new result: [1-1, 1-1] = [0, 0]

In [204]:

```
a = np.array([10, 15, 25, 5])  
diff = np.diff(a, n=2)  
  
print(diff)  
[ 5 -30]
```

In [205]: a = np.array([1, 2, 3, 4])

```
diff = np.diff(a, n=2)  
  
print(diff)
```

```
[0 0]
```

## NumPy LCM Lowest Common Multiple

The Lowest Common Multiple is the smallest number that is a common multiple of two numbers.

In [206]:

```
num1 = 4  
num2 = 6  
  
x = np.lcm(num1, num2)  
  
print(x)
```

```
12
```

The **reduce()** method will use the **ufunc**, in this case the **lcm()** function, on each element, and reduce the array by one dimension.

In [207]:

```
a = np.array([3, 6, 9])  
  
x = np.lcm.reduce(a)  
  
print(x)
```

```
18
```

Returns: 18 because that is the lowest common multiple of all three numbers ( $36=18$ ,  $63=18$  and  $9*2=18$ ).

## NumPy GCD Greatest Common Denominator

In [208]:

```
num1 = 6  
num2 = 9  
  
x = np.gcd(num1, num2)  
  
print(x)
```

Returns: 3 because that is the highest number both numbers can be divided by ( $6/3=2$  and  $9/3=3$ ).

The **reduce()** method will use the ufunc, in this case the **gcd()** function, on each element, and reduce the array by one dimension.

```
In [209]: a = np.array([20, 8, 32, 36, 16])
x = np.gcd.reduce(a)
print(x)
```

4

Returns: 4 because that is the highest number all values can be divided by.

## NumPy Trigonometric Functions

NumPy provides the ufuncs **sin()**, **cos()** and **tan()** that take values in radians and produce the corresponding **sin**, **cos** and **tan** values.

```
In [210]: x = np.sin(np.pi/2)
print(x)
```

1.0

```
In [211]: pi2 = np.array([np.pi/2, np.pi/3, np.pi/4, np.pi/5])
x = np.sin(pi2)
print(x)
```

[1. 0.8660254 0.70710678 0.58778525]

```
In [212]: # Convert Degrees Into Radians
a = np.array([90, 180, 270, 360])
x = np.deg2rad(a)
print(x)
```

[1.57079633 3.14159265 4.71238898 6.28318531]

```
In [213]: # Radians to Degrees
a = np.array([90, 180, 270, 360])
x = np.rad2deg(a)
print(x)
```

[ 5156.62015618 10313.24031235 15469.86046853 20626.48062471]

**NumPy** provides *ufuncs* `arcsin()`, `arccos()` and `arctan()` that produce radian values for corresponding *sin*, *cos* and *tan* values given.

```
In [214]: x = np.arcsin(1.0)
```

```
print(x)
```

```
1.5707963267948966
```

```
In [215]: a = np.array([1, -1, 0.1])
```

```
x = np.arccos(a)
```

```
print(x)
```

```
[0. 3.14159265 1.47062891]
```

```
In [216]: a = np.array([1, -1, 0.1])
```

```
x = np.arctan(a)
```

```
print(x)
```

```
[ 0.78539816 -0.78539816  0.09966865]
```

```
In [217]: # Hypotenues
```

```
base = 3
```

```
perp = 4
```

```
x = np.hypot(base, perp)
```

```
print(x)
```

```
5.0
```

## NumPy Hyperbolic Functions

**NumPy** provides the *ufuncs* `sinh()`, `cosh()` and `tanh()` that take values in radians and produce the corresponding *sinh*, *cosh* and *tanh* values.

```
In [218]: x = np.sinh(np.pi/2)
```

```
x
```

```
Out[218]: 2.3012989023072947
```

```
In [219]: x = np.cosh(np.pi/2)
```

```
x
```

```
Out[219]: 2.5091784786580567
```

```
In [220]: x = np.tanh(np.pi/2)
```

```
x
```

```
Out[220]: 0.9171523356672744
```

## Finding Angles

Numpy provides **ufuncs *arcsinh()*, *arccosh()* and *arctanh()*** that produce radian values for corresponding *sinh*, *cosh* and *tanh* values given.

```
In [221]: x = np.arcsinh(1.0)
```

```
print(x)
```

```
0.881373587019543
```

## NumPy Set Operations

```
In [222]: ab = np.array([1, 1, 1, 2, 3, 4, 5, 5, 6, 7])
```

```
x = np.unique(ab)
```

```
print(x)
```

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

```
In [223]: # Finding Union
```

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

```
x = np.union1d(a, b)
```

```
print(x)
```

```
[1 2 3 4 5 6]
```

```
In [224]: # Finding Union
```

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

```
x = np.intersect1d(a, b, assume_unique=True)
```

```
print(x)
```

```
[3 4]
```

```
In [225]: a = np.array([1, 2, 3, 4])
b = np.array([3, 4, 5, 6])

x = np.setdiff1d(a, b, assume_unique=True)

print(x)

[1 2]
```

```
In [226]: a = np.array([1, 2, 3, 4])
b = np.array([3, 4, 5, 6])

x = np.setxor1d(a, b, assume_unique=True)

print(x)

[1 2 5 6]
```

# Download Machine Learning Study Material:

