NumPy

Numpy :-

- Numpy is python library.
- Numpy is used for working with arrays.
- Numpy is short for "Numerical Python".
- It has functions for working in domain of linear algebra, fourier transform, and matrices.

Why Use Numpy :-

- Numpy provides a powerful array object called "ndarray", which is much faster (up to 50x) than python lists. It's essential for data science due to its speed, efficiency, and built-in functions for easy array manipulation

Why Numpy is faster than Lists?

- Numpy arrays use contiguous memory, enabling faster access (locality of refrence) and are optimized for modern CPUs, making them much faster than python lists.

Which Language is Numpy written In?

- Numpy is mostly written in C & C++ for performance with python interface for ease of use.

Installation of Numpy :-

* pip install numpy
* import numpy as np
* np.__version__ ——→ (To check version of numpy)

Create a Numpy ndarray Object :-

- Numpy is used to work with arrays & this array object called "ndarray".
- It is created by using the array() function.

Example:-

```
import numpy as np
arr = np.array([1,2,3,4,5])
print(arr)
print(type(arr))
```

type :- This built in function tells us the type of object passed to it. like arr is numpy.ndarray type

- To create an ndarray, we can pass list, tuple or any array-like object into the array() method, and it will be converted into an ndarray.

Example:-

```
import numpy as np
arr = np.array((1,2,3,4,5))
print(arr)
```

Here tuple is passed

Dimensions in Arrays :-

- A dimension in array is one level of array depth (nested arrays)

0-D Arrays:-

- 0-D arrays, or Scalars, are the elements in an array, Each value in an array is a 0-D array.

Example:-

Q. create a 0-D array with value 42

→
```
import numpy as np
arr = np.array(45)
print(arr)
```

ans → 45 (After compilation the code)

1-D Arrays:-

- An array thas has 0-D arrays as its elements is called uni-dimensional or 1-D array.
- These are the most common and basic arrays.

Example:-

Q. create a 1-D array containing the values. 1,2,3,4,5.

→
```
import numpy as np
arr = np.array([1,2,3,4,5])
print(arr)
```

ans → [1,2,3,4,5] (After compilation)

2-D Arrays:-

- An array that has 1-D arrays as its elements is called a 2-D array.
- These offen used to represent matrix or 2nd order tenso

- Numpy has a whole module dedicated towards matrix operations called "numpy.mat"

Example

Q. create a 2-D array containing two arrays with the values 1,2,3 & 4,5,6.

→
```
import numpy as np
arr = np.array([[1,2,3],[4,5,6]])
print(arr)
```

→
```
[[1,2,3].
 [4,5,6]]
```
← (After compilation)

3-D Arrays :-

Q. An array that has 2-D arrays, both containing two arrays with the values 1,2,3 & 4,5,6. create such 3-D array containing this.

→
```
import numpy as np
arr = np.array([[[1,2,3],[4,5,6]],[[1,2,3],[4,5,6]]])
print(arr)
```

→
```
[[[1,2,3]
  [4,5,6]]
 [[1,2,3]
  [4,5,6]]]
```
← (Ans after compilation)

- An array that has 2-D arrays (matrices) as its elements is called 3-D arrays.
- These are often used to represent a 3rd order tensor.

Check Number of Dimensions:-

Numpy arrays provides the ndim attribute that returns an integer that tells us how many dimensions the array have.

Example:- Q. check How many dimensions the arrays have.

→
```
import numpy as np
a = np.array(42)
b = np.array([1,2,3,4,5])
c = np.array([[1,2,3],[4,5,6]])
print(a.ndim)
print(b.ndim)
print(c.ndim)
```

→
```
0
1
2
```
→ (Ans. after compilation)

Higher Dimensional Arrays:-

- An array can have any number of dimensions. When the array is created, you can define the number of dimension by using the ndmin argument.

Example:- Q. Create an array with 5 dimensions & verify that it has 5 dimensions.

→
```
import numpy as np
arr = np.array([1,2,3,4], ndmin=5)
Print(arr)
print('number of dimensions :', arr.ndim)
```

→ [[[[[1,2,3,4]]]]]
number of dimensions : 5 → (Ans. after comp)

- In this array the innermost dimension (5th dim) has ~~4th element~~ 4 element, the 4th dim has 1 element that is the vector, the 3rd dim has 1 element that is the matrix with the vector, the 2nd dim has 1 element that is 3D array and 1st dim has 1 element that is a 4D array.

Numpy Array Indexing :-

Access Array elements :-

- Array indexing is the same as accessing an array element.
- you can access an array element by referring to its index number.
- The indexes in Numpy array starts with 0, meaning that first element has index 0, and second has index 1 etc.

Example :- Q. get the 1st element of following array :-

```
import numpy as np
arr = np.array([1,2,3,4])
print(arr[0])
```

→ 1 ——— (ANS. After comp)

Q.2. Get the 3rd & 4th element from following array & add them.

→ import numpy as np
arr = np.array([1,2,3,4])
print(arr[2] + arr[3])

→ 7 → (ANS. after. comp).

Accessing 2-D Arrays:-

- In a 2-D array, use array[dimension, index] to get an element.

- Think of it like a table:-
  - Dimension = row number
  - Index = Coloumn number

for example, array[1,2] means:
2nd row (dimension 1), 3rd coloumn (index 2)

Example:-

Q. Access the element on 2nd row, 5th coloumn.

→ import numpy as np
arr = np.array([[1,2,3,4,5],[6,7,8,9,10]])
print('5th element on 2nd row:', arr[1,4])

→ 5th element on 2nd row : 10

Access 3-D arrays:-

- To access element in a 3-D array, use array[dim1, dim2, index]

- Think of it like a stack of tables:

dim1 = which table (depth)
dim2 = row in that table
dim3 = column in that row.

e.g. array [0, 1, 2] means,
1st table → 2nd row → 3rd column.

Example:-

Q. Access the third element of the second array of the first array.

```
import numpy as np
arr = np.array([[[1,2,3],[4,5,6]],[[7,8,9],[10,11,12]]])
print(arr[0,1,2])
```

→ 6 ans ——→ (After comp)

Negative indexing :-

- Use negative indexing to access an array from the end.

Example:- Q. print the last element from 2nd dim.

```
import numpy as np
arr = np.array([[1,2,3,4,5],[6,7,8,9,10]])
print('Last element from 2nd dim:', arr[1,-1])
```

→ Last element from 2nd dim : 10 ——(ANS after comp)

Numpy Array Slicing:-

- Slicing in python means taking element from one given index to another given index.

- We pass slice instead of index like this [start:end].
- We can also define the step, like this [start:end:step].
- If we don't pass start its considered 0
- If we don't pass end its considered length of array in that dimension.
- If we don't pass step its considered 1

Example:-

Q1. Slice the elements from index 1 to index 5 from the following array.

→

```
import numpy as np
arr = np.array([1,2,3,4,5,6,7])
print(arr[1:5])
```

→ [2,3,4,5] ——— (Ans. after. comp)

Q.2 Slice the element from index 4 to the end of the array.

→ * same array *

```
print(arr[4:])
```

→ [5,6,7] —— (Ans. after comp.)

Q.3. Slice elements from the beginning to index 4 (not included)

→ * same array *

```
print(arr[ :4])
```

→ [1,2,3,4] —— (ans. after. comp.)

Negative slicing :-

- Use minus operator to refer to an index from the end.

Example :-

Q1. Slice from the index 3 from the end to index 1 from the end.

→

```
import numpy as np
arr = np.array([1,2,3,4,5,6,7])
print(arr[-3:-1])
```

→ [5,6] —— (Ans. after. Comp)

Step :-

- Use the step value to determine the step of the slicing.

Example :-

Q1. Retuen every other element from index 1 to index 5:

→

```
import numpy as np
arr = np.array([1,2,3,4,5,6,7])
print(arr[1:5:2])
```

→ [2,4] —— (Ans. after. comp.)

Q2. Return every other element from the entire array:

→ * same array *

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

→ [1, 3, 5, 7] —— (Ans. after comp.)

② Slicing 2-D Arrays:-

Q.1. from the second element, slice element from index 1 to index 4 (not included)

→

```
import numpy as np
arr = np.array([[1,2,3,4,5],[6,7,8,9,10]])
print(arr[1, 1:4])
```

→ [7, 8, 9] —— (Ans. after comp.)

Q.2. from both elements, return index 2:

→ * same array *

```
print(arr[0:2, 2])
```

→ [3, 8] —— (Ans. after comp)

Q.3 from both elements, slice index 1 to index 4 (not included) this will return 2-D array.

→ * same array *

```
print(arr[0:2, 1:4])
```

→ [[2, 3, 4], —— (Ans. after comp)
[7, 8, 9]]

Numpy Data Types :-

- Below is a list of all data types in numpy and the characters used to represent them.

i -	Integer
b -	boolean
u -	unsigned integer
f -	float
c -	complex float
m -	timedelta
M -	datetime
O -	object
S -	string
U -	unicode string
V -	fixed chucks of memory for other type (void)

Example :-

Q1. Get the data type of an array object

→

```
import numpy as np
arr = np.array([1,2,3,4])
print(arr.dtype)
```

→ int64 ——— (Ans. after comp)

⑪ Creating Arrays with a Defined Data Type:-

Q1) create an array with data type string.

→

```
import numpy as np
arr = np.array([1,2,3,4], dtype='S')
print(arr)
print(arr.dtype)
```

→ `[b'1' b'2' b'3' b'4']` —— (Ans. after comp)

|S1

- each number becomes a string & stored stored as bytes b'1', b'2' etc.
- dtype becomes |S1, which means,
  - S = string (byte string)
  - 1 = length of each string is 1 character.

for i, u, f, S & U we can define size as well.

Q.2 create an array with data type 4 bytes integer.

```
import numpy as np
arr = np.array([1,2,3,4], dtype='i4')
print(arr)
print(arr.dtype)
```

→ `[1,2,3,4]` —— (Ans. after comp)

int32

Converting Data Type on Existing Arrays!

- Use the .astype() method to create a copy of an array with a new data type.

- you can specify the type as,
  - A string : 'f' (float), 'i' (int) etc
  - A type : float, int etc.

Example:-

Q1. Change data type from float to integer by using 'i' as parameter.

→

```
import numpy as np
arr = arr.astype('i')
print(arr)
print
arr = np.array([1.1, 2.1, 3.1])
newarr = arr.astype('i')
print(newarr)
print(newarr.dtype)
```

→ [1, 2, 3] ——— (Ans. after comp)
int32

Q2. Do the above q same but use 'int' instead 'i'

→

```
import numpy as np
arr = np.array([1.1, 2.1, 3.1])
newarr = arr.astype(int)
print(newarr)
print(newarr.dtype)
```

→ [1,2,3]
int64

Q.3 change data type from integer to boolean.

→

```
import numpy as np
arr = np.array ([1,0,3])
newarr = arr.astype (bool)
print (new arr)
print (new arr.dtype)
```

→ [True False True] ——— (Ans. after comp)
bool

Numpy Array Copy Vs View:-

Copy :-

- creates a new array with its own data.
- changes in the copy do not affect the original arrays, and vice versa.

View :-

- creates a linked view of the original array
- changes in the view do affect the original arrays, and vice versa.

Example:-

Q1. Make a copy, change the original array & display both arrays.

→

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5])
x = arr.copy()
arr[0] = 42
print(arr)
print(x)
```

→ [42, 2, 3, 4, 5] — (Ans. after comp.)
[1, 2, 3, 4, 5]

Q2. Make a view, change the original array, and display both arrays.

→

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5])
x = arr.view()
arr[0] = 42
print(arr)
print(x)
```

→ [42, 2, 3, 4, 5] — (Ans. after comp.)
[42, 2, 3, 4, 5]

❸ Check if Array owns its Data:-

Use the .base attribute:-

- If .base is none, the array owns its data (i.e. it's copy)
- If .base is not None, it's a view of another array

Example:-

Q1. print the value of the base attribute to check if an array owns it's data or Not.

→

```
import numpy as np
arr = np.array([1,2,3,4,5])
x = arr.copy()
y = arr.view()
print(x.base)
print(y.base)
```

→ None ——— (Ans. after comp)
[1,2,3,4,5]

Numpy Array Shape:-

- The shape of an array is the number of element in each dimension.

- Use the .shape attribute to get the dimensions of a Numpy array.

- It returns a tuple showing the size along each dimensions.

Example:-

```
import numpy as np
arr = np.array([[1,2,3,4],[5,6,7,8]])
print(arr.shape)
```

→ (2,4)

which means that the array has 2 dimension, where the 1st dimension has 2 element & 2nd has 4 elements.

Q2. create a array with 5 dimensions using ndmin using a veetor with values 1, 2, 3, 4 & verify that the last dimension has value 4

→

```
import numpy as np
arr = np.array([1,2,3,4], ndmin=5)
print(arr)
print('shape of array:', arr.shape)
```

→ `[[[[[1 2 3 4]]]]]`
shape of array : (1,1,1,1,4)

What Does the shape Tuple Represent?

- The shape tuple shows how many elements exist along each dimension of the array

- for shape = (1,1,1,1,4)

1st dimension → 1 element
2nd dimension → 1 element
3rd dimension → 1 element
4th dimension → 1 element
5th dimension → 4 element

So, the value 4 at index 4 in the shape tuple tells us that the 5th dimension has 4 element.

- Summary, Each index in .shape corresponds to a dimension, and its value tells how many elements exist along that dimension.

Numpy Array Reshaping :-

- Reshaping means changing the structure of an array
- Its dimension and how elements are arranged.
- The shape defines hoow many elements are in each dimensions

With reshaping, you can;
- Add or remove dimensions.
- Change the number of elements per dimension. (total elements must stay the same)

Example:-

Q.1. Convert the following 1-D array with 12 elements into a 2-D array. The outmost dimension will have 4 arrays, each with 3 elements:

→

```
import numpy as np
arr = np.array([1,2,3,4,5,6,7,8,9,10,11,12])
newarr = arr.reshape(4,3)
print(newarr)
```

→
```
[[ 1, 2, 3]
 [ 4, 5, 6]
 [ 7, 8, 9]
 [10,11,12]]
```
(Ans. after comp)

Q.2. Convert the following 1-D array with 12 elements into a 3-D array. The outermost dimension will have 2 arrays that contains 3 arrays, each with 2 elements.

→

```
import numpy as np
arr = np.array([1,2,3,4,5,6,7,8,9,10,11,12])
newarr = arr.reshape(2,3,2)
print(newarr)
```

→
```
[[[1, 2]
  [3, 4]
  [5, 6]],
 [[7, 8]
  [9, 10]
  [11,12]]]
```
—— (Ans. after Comp)

Can we reshape into Any ~~array~~ shape?

Yes — but only if the total number of elements stays the same

<u>for Example</u>:-

- A 1-D array with 8 elements can be reshaped to (2,4) or (4,2)
- It cannot be reshaped to (3,3) because that needs 9 elements

<u>Rule</u>:-

original size = New shape size

(np.reshape() will throw an error if the sizes don't match).

⑧ Returns Copy or View!-

Q.1. check if the returned array is a copy or a view

→

```
import numpy as np
arr = np.array([1,2,3,4,5,6,7,8])
print(arr.reshape(2,4).base)
```

→ [1,2,3,4,5,6,7,8] —— (Ans. after comp)

- The example above returns the original array, so its a view.

⑨ Unknown Dimension :-

- You can let Numpy automatically calculate one dimension by using -1

- This is useful when you are unsure of the exact size but know the total number of elements.

Example

Q.1 convert 1-D array with 8 elements to 3D array with 2x2 elements.

→

```
import numpy as np
arr = np.array([1,2,3,4,5,6,7,8])
newarr = arr.reshape(2, 2, -1)
print(newarr)
```

→
```
[[[1,2]
  [3,4]]
 [[5,6]
  [7,8]]]
```
—— (Ans. after. Comp)

- We cannot pass -1 to more than one dimension.

⑥ Flattening the array:-

- Flattening the array means converting multidimensional array into a 1D array.
- We can use reshape(-1) to do this.

Example:-

Q.1. Convert the array into a 1D array.

→

```
import numpy as np
arr = np.array([[1,2,3],[4,5,6]])
newarr = arr.reshape(-1)
print(newarr)
```

→ [1,2,3,4,5,6] —— (Ans. after comp)

Note:-

Numpy offers many functions for changing array shapes & rearranging elements.

- Shap-changing:-
  - flatten(), ravel() —— convert arrays to 1-D
- Rearranging elements
  - rot90() —— rotate array 90° (degrees)
  - flip(), fliplr(), flipud() — flips arrays in various directions.

These are part of intermediate to advanced Numpy operations.

Iterating Arrays :-

- Iteration means accessing elements one by one
- for 1-D arrays, a simple python for loop will go through each element in sequence.

Example :-

Q1. Iterate on the elements of the following 1-D array.

→

```
import numpy as np
arr = np.array([1,2,3])
for x in arr:
  print(x)
```

→ 1
2 ——— (ANS. after comp)
3

Q2. Iterate on the elements of the following 2-D array.

→

```
import numpy as np
arr = np.array([[1,2,3],[4,5,6]])
for x in arr
  print(x)
```

→ [1, 2, 3] ——— (ANS. after comp)
[4, 5, 6]

If we iterate on a n-D array it will go through n-1th dimension one by one.

To return the actual values, the scalars, we have to iterate the arrays in each dimension.

Q3. Iterate on each scalar element of the 2-D array.

→

```
import numpy as np
arr = np.array([[1,2,3],[4,5,6]])
for x in arr:
    for y in x:
        print(y)
```

→

1
2
3 ——— (ANS. after comp)
4
5
6

Q4. Iterate on the elements of the following 3-D arrays.

→

```
import numpy as np
arr = np.array([[[1,2,3],[4,5,6]],[[7,8,9],[10,11,12]]])
for x in arr:
    print(x)
```

To return the actual values, the scalars, we have to iterate the arrays in each dimension.

Q5. Iterate down to the scalars!

→

```
import numpy as np
arr = np.array([[[1,2,3],[4,5,6]],[[7,8,9],[10,11,12]]])
for x in arr:
    for y in x:
        for z in y:
            print(z)
```

→

```
1
2
3
4
!
!
12
```

(Ans. after comp)

Iterating Arrays Using "nditer()" :-

- nditer() is a Numpy helper function for simple to advanced iteration.

- It allows you to loop through each scalar element of an array without writing multiple nested loops — very useful for High-dimensional arrays.

Example

Q1. Iterate through the following 3-D array.

→

```
import numpy as np
arr = np.array([[[1,2],[3,4]], [[5,6],[7,8]]])
for x in np.nditer(arr):
    print(x)
```

→ 1
2
.
.
8 ——— (ANS. after comp)

Iterating Array with Different Data Types:—

- When using nditer(), you can change the datatype of elements during iteration by using the op_dtypes argument.

- Since Numpy doesn't change the data type in-place, it uses a temporary buffer for the conversion. to enable this, pass flags = ['buffered'].

01. Iterate through the array as a string:

```
import numpy as np
arr = np.array([1,2,3])
for x in np.nditer(arr, flags=['buffered'], op_dtypes=['S']):
    print(x)
```

→ b'1'
b'2' ——— (ANS. after comp)
b'3'

We can use filtering and followed by iteration.

⑦ Iterating with Different Step size:-

Q1. Iterate through every scalar elements of the 2D array skipping 1 element

⟶

```
import numpy as np
arr = np.array([[1,2,3,4],[5,6,7,8]])
for x in np.nditer(arr[:, ::2]):
    print(x)
```

⟶
```
1
3
5
7
```
(Ans. after comp)

⑧ Enumerated iteration Using ndenumerate() :-

- Enumeration means iterating with the index of each element.

- ndenumerate() lets you loop through an array and get both.
    * The index (as a tuple for multi-dimensional arrays)
    * The value at that index

Q1. Enumerate on following 1-D array's elements?

⟶

```
import numpy as np
arr = np.array([1,2,3])
for idx, x in np.ndenumerate(arr):
    print(idx,x)
```

⟶

(0,) 1
(1,) 2 ——— (Ans. after comp)
(2,) 3

Q2. Enumerate on following 2D array's elements,

→

```
import numpy as np
arr = np.array([[1,2,3,4],[5,6,7,8]])
for idx, x in np.ndenumerate(arr):
    print(idx, x)
```

→
(0,0) 1
(0,1) 2
(0,2) 3
(0,3) 4
(1,0) 5
(1,1) 6
(1,2) 7
(1,3) 8

Numpy Joining Array :-

- Joining means combining two or more arrays into a single array.

- Unlike SQL (where joins are based on keys), Numpy joins array along an axis

- Use np.concatenate():
  - pass the arrays as a sequence (list or tuple)
  - Specify the axis (default is 0)

Example :-

Q1. Join two arrays.

→

```
import numpy as np
arr1 = np.array([1,2,3])
arr2 = np.array([4,5,6])
arr = np.concatenate((arr1, arr2))
print(arr)
```

→ [1,2,3,4,5,6] ——— (ANS. after comp)

Q2. Join two 2-D arrays along rows (axis=1):

→

```
import numpy as np
arr1 = np.array([[1,2],[3,4]])
arr2 = np.array([[5,6],[7,8]])
arr = np.concatenate((arr1, arr2), axis=1)
print(arr)
```

→ [[1,2,5,6]
[3,4,7,8]] ——— (ANS. after comp)

Joining Arrays Using Stack Functions :-

- Stacking is like concatenation, but it adds a new axis.

- This means two 1-D arrays can be combined along a new dimension — placing them on top of each other or side by side.

Use np.stack():

- pass the arrays as a sequence
- Specify axis (default = 0 → stack vertically)

Example:-

```
import numpy as np
arr1 = np.array([1,2,3])
arr2 = np.array([4,5,6])
arr = np.stack((arr1, arr2), axis=1)
print(arr)
```

→ [[1,4]
[2,5]
[3,6]]

(ANS. after comp)

Stacking along rows:-

- Numpy provides a helper function: hstack() to stack along rows.

Example:-

```
import numpy as np
arr1 = np.array([1,2,3])
arr2 = np.array([4,5,6])
arr = np.hstack((arr1,arr2))
print(arr)
```

→ [1,2,3,4,5,6] (ANS. after comp)

① Stacking along Columns :-

- Numpy provide a helper function, Vstack(), to stack along columns.

Example:-

```
import numpy as np
arr1 = np.array([1,2,3])
arr2 = np.array([4,5,6])
arr = np.vstack((arr1, arr2))
print(arr)
```

→ `[[1,2,3]`
`[4,5,6]]` —— (Ans. after comp)

② Stacking along Height (depth) :-

- Numpy provides a helper function: dstack(), to stack along height, which is the same as depth

Example:-

```
import numpy as np.
arr1 = np.array([1,2,3])
arr2 = np.array([4,5,6])
arr = np.dstack((arr1, arr2))
print(arr)
```

→ `[[[1,4]`
`[2,5]`
`[3,6]]]` —— (Ans. after comp)

Numpy Splitting array :-

- Splitting in numpy is the opposite of joining

- Instead of combining arrays, It devides one array into multiple parts.

- The function np.array_split() is used for this, where you provide the array and how many parts you want.

Example:-

```
import numpy as np
arr = np.array([1,2,3,4,5,6])
newarr = np.array_split(arr,3)
print(newarr)
```

⟶

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

The retuen value is a list containing three arrays.

If the array has less elements than required, it will adjust from the end accordingly.

Example:-

Q1 split the array in 4 parts

⟶

```
import numpy as np
arr = np.array([1,2,3,4,5,6])
newarr = np.array_split(arr, 4)
print(newarr)
```

→ [array([1,2]), array([3,4]), array([5]), array([6])]

split() requires the array to be evenly divisible into parts, while array_split() can hadle uneven splits by adjusting element distribution.

Split into Arrays :-

- The array_split() method returns a list of Numpy arrays.
- If you split into 3 parts, you can access each part by index. e.g. result[0], result[1], result[2].

Q1. Access the splitted arrays.

→

```
import numpy as np
arr = np.array([1,2,3,4,5,6])
newarr = np.array_split(arr,3)
print(newarr[0])
print(newarr[1])
print(newarr[2])
```

→ [1,2]
[3,4] ——— (ANS. after comp)
[5,6]

Splitting 2-D Arrays:-

- It works the same way as 1-D array - use array-split(array, num-splits) to divide it into the desired number of subarrays.

Example:-

Q1. Split the 2-D array into three 2-D arrays.

→

```
import numpy as np
arr = np.array([[1,2],[3,4],[5,6],[7,8],[9,10],[11,12]])
newarr = np.array_split(arr, 3)
print(newarr)
```

→

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

The example above returns three 2-D arrays.

In addition, you can specify which axis you want to do the split around.

The example below also returns three 2-D arrays, but they are split along the column (axis=1)

Example:-

Q1. Split the 2-D array into three 2-D arrays along columns.

→

```
import numpy as np
arr = np.array([[1,2,3],[4,5,6],[7,8,9],[10,11,12],[13,14,15],[16,17,18]])
newarr = np.array_split(arr,3, axis=1)
print(newarr)
```

→ [array([[1],
[4],
[7],
[10],
[13],
[16]]),
array([[2],
[5],
[8],
[11],
[14],
[17]])
array([[3],
[6],
[9],
[12],
[15],
[18]])]

An alternate solution is using hsplit() opposite of hstack(). & also similar function exist

~~∴ np.array~~

1) ∴ np.hsplit(arr,3) —— (along columns
2) ∴ vsplit() —— (split vertically along rows)
3) dsplit() —— (split along depth (3-Darrays)

Numpy Searching Arrays :-

- In Numpy, you can use where() to find the indexes of elements that match a condition.

- Works on any array (sorted or unsorted)
- Returns the indices of elements that satisfy a condition. (tuple of arrays)

Example:-

```
import numpy as np
arr = np.array([4,7,4,9,4,10])
# find all position where value is 4
idx = np.where(arr==4)
idx1 = np.where(arr%2==0)
idx2 = np.where(arr%2==1)
print(idx)      # (array([0,2,4]),)
print(arr[idx]) # [4,4,4]
print(idx1)     # (array([0,2,4,5]),)
print(idx2)     # (array([1,3,]))
```

② np.searchsorted() → find insert position.

- Works only on sorted arrays.
- Returns the index where a value should be inserted to keep the array sorted.
- Can insert left or right of duplicate

Example:-

```
import numpy as np
arr = np.array([4,7,9,12])   # must be sorted.
x = np.searchsorted(arr, 7)
pos_left = np.searchsorted(arr, 7, side = 'left')
pos_right = np.searchsorted(arr, 7, side = 'right')
print(x)          # 1 → before 7 (default)
print(pos_left)   # 1 → before 7
print(pos_right)  # 2 → after 7
```

Numpy Array Sorting:-

- purpose → Sorts elements in ascending order, either numerically or alphabetically, depending on the array's data type.

- Function used → np.sort() — returns a sorted copy, leaving the original array unchanged.

- Works with → 1D arrays — sort all elements
  2D arrays — sort each ~~element~~ row independently (default axis)

- Example →

```
import numpy as np
arr1d = np.array([3,2,0,1])
print(np.sort(arr1d))   # [0,1,2,3]
arr2d = np.array([[3,2,4],[5,0,1]])
print(np.sort(arr2d))   # [[2,3,4]
                        #  [0,1,5]]
```

Filtering Numpy Arrays:-

- Filtering means extracting specific elements from an array based on condition.

- In Numpy, this is done using boolean indexing — supplying a boolean array (True/False values) to select elements.

Example

```
import numpy as np.
arr = np.array ([41, 42, 43, 44])
filter_arr = [True, False, True, False]
newarr = arr[filter_arr]
print (new arr)        # output : [41, 43]

# only the elements at indices where the filter is
  True are included.
```

Building filter dynamically - Based on condition:-

```
arr = np.array ([41, 42, 43, 44])
filter_arr = [x > 42 for x in arr] # [false, false, True True]
newarr = arr [filter_arr]
print (newarr)  # output : [43, 44]

# you can generate the boolean filter using
any logic.
```

⊖ Even Simpler - Direct Boolean Masking:

```
arr = np.array([1,2,3,4,5,6,7])
newarr = arr[arr%2 == 0]
print(newarr) # output [2, 4, 6]
```

Here, `arr%2 == 0` creates a boolean mask directly, without needing an explicit list.

Key Takeaways:-

Boolean indexing ⟶ Use a True/False mask array to select elements from numpy array.

Manual mask list ⟶ Build a list of True/False and apply it to the array.

Conditional mask shortcut ⟶ Use expression like `arr > value`, `arr%2 == 0`, etc, to create the mask.

Random Number in Numpy :-

from Numpy import random

Random Integers :-

Single :- random.randint(100) ⟶ int between (0-99)

Array :- random.randint (100, size=(3,5))

Random Floats :-

Single :- random.rand() ⟶ float between 0-1

Array :- random.rand(3,5)

Random choice from List :-

Single :- random.choice ([3,5,7,9])

Array :- random.choice ([3,5,7,9], size=(3,5))

Numpy Random Distribution :-

① Data Distribution ⟶ shows all posible values + how often they occur.

Random Distribution in Numpy ⟶ Generate random samples following a given probability.

random.choice()

- Picks random values from a given list/array.
- Can assign probabilities with p parameter.
- Can create 1D or multi-dimensional arrays using size.

Example

```
from numpy import random
x = random.choice([3,5,7,9], p=[0.1,0.3,0.6,0.0], size=100)
    # for 1D: 100 random values.
y = random.choice([3,5,7,9], p=[0.1,0.3,0.6,0.0], size=(3,5))
```

Here, for all element in array,

a- 3 → 10% chance.
5 → 30% chance.
7 → 60% chance
9 → 0% chance. (won't appear)

Numpy Random Permutations (shuffle() & permutation()).-

- Permutation = any rearrangement of elements.
(e.g. [3,2,1] is a permutation of [1,2,3]).

- Numpy's random module offers two main tools!-

- Shuffle():- Randomly shuffles an array in-place, modifying the original array.

- Permutation:- Returns a shuffled copy of the array, leaving the original unchanged.

Examples (short & sweet):-

```
from numpy import random
import numpy as np.

arr = np.array([1,2,3,4,5])
random.shuffle(arr)          # shuffles arr in-place.
print(arr)
```

→ [2,4,1,5,3] — (Ans after comp)

```
print(random.permutation(arr)) # Returns a shuffled
                                 Copy
```

→ [3,2,1,5,4] — (Ans after comp)

Numpy Normal Distribution:-

- Normal (Gaussian) Distribution = bell curve.
- Centered around a mean (average), spread controlled by standard deviation (std)
- Used to model natural variations (e.g. heights, test scores, measurement error)

⊛ Function:-

```
random.normal(loc=0.0, scale=1.0, size=None)
```

loc → mean (centre of curve)
scale → standard Deviation (spread of values)
size → shape of output array.

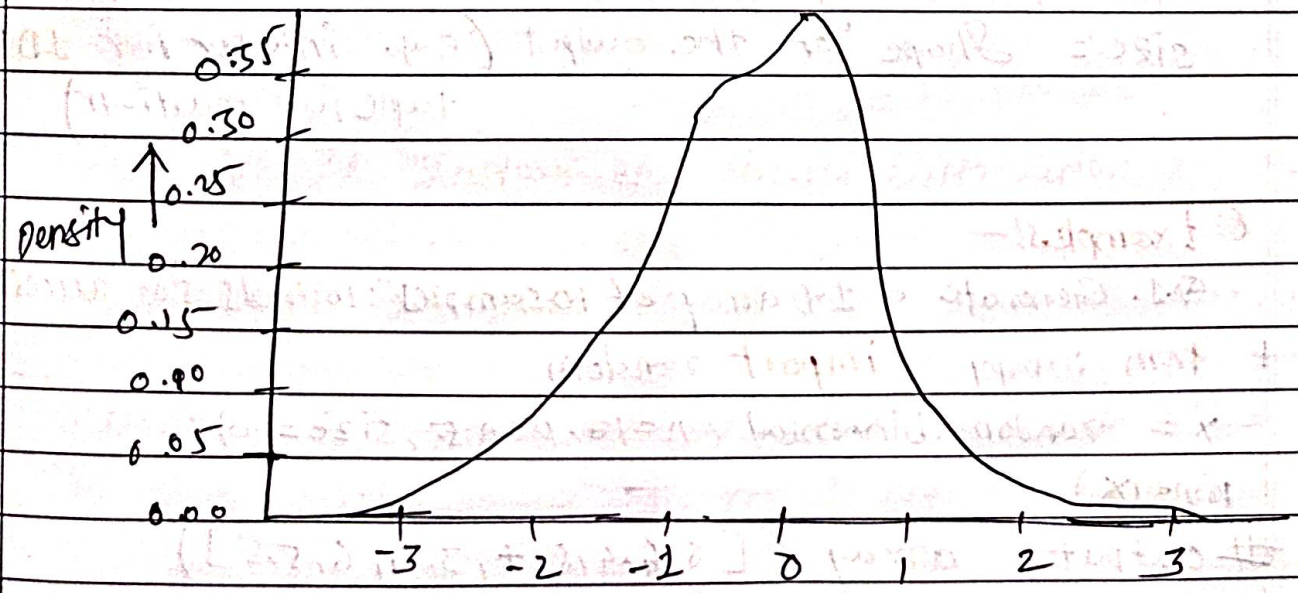
Examples:-

```
from numpy import random
# single random value (mean=0, std=1)
x = random.normal()
# 2x3 array with default mean=0, std=1
x = random.normal(size=(2,3))
# 2x3 array with mean=1; std=2
x = random.normal(loc=1, scale=2, size=(2,3))
```

Visualization:-

```
import matplotlib.pyplot as plt
import seaborn as sns
from numpy import random

data = random.normal(loc=0, scale=1, size=1000)
sns.displot(data, kind="kde")  # bell curve
plt.show()
```



-key points:-

mean (loc) ⟶ shifts curve left/right
std Dev (scale.) ⟶ controls curve width
size ⟶ controls how many values are generated.

Numpy Binomial Distribution:-

- A discrete probability distribution modeling the number of successes in a repeated independent trials (each trial has a binary outcome e.g. success/failure)
Example :- coin flips, pas/fail tets.

Function Syntax:-

random.binomial (n, p, size=None)

n = Number of trials (integer)
p = probability of success in each trial ($0 \leq p \leq 1$)
size = Shape of the output (e.g. integer for 1D array, tuple for multi-D)

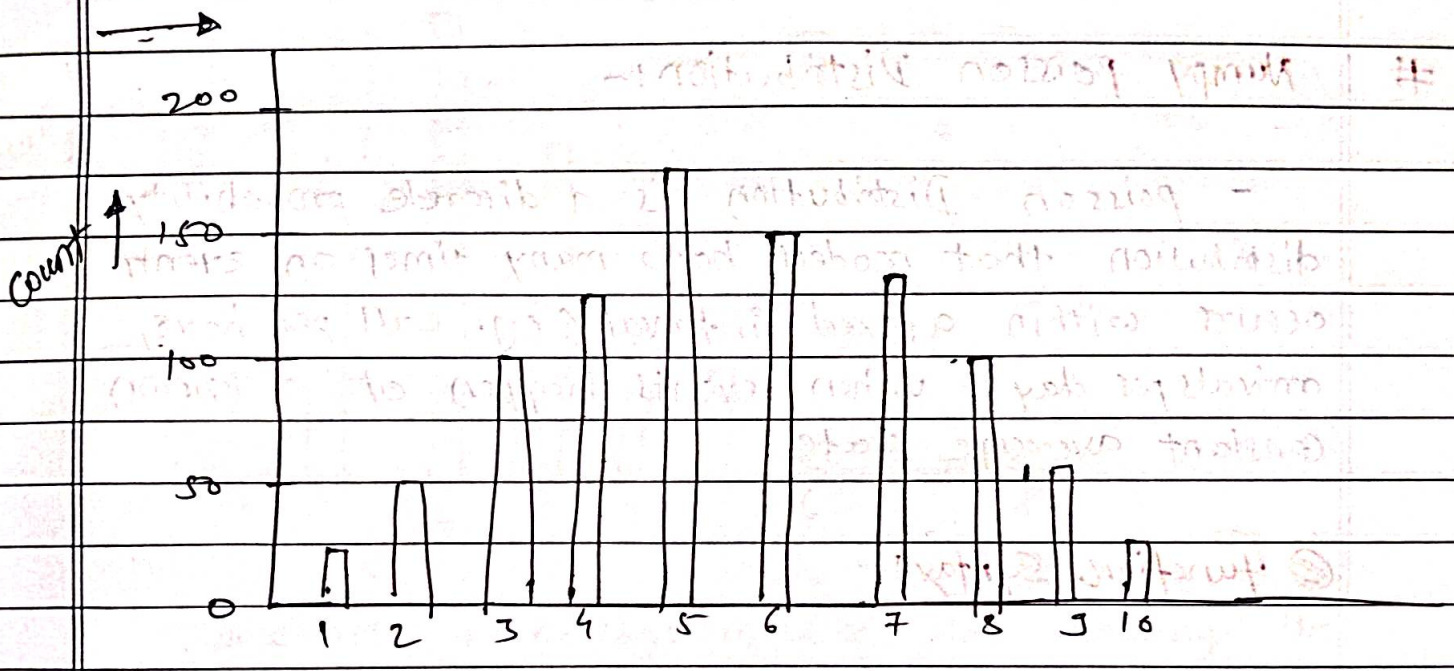
Examples:-

Q.1. Generate a 1-D array of 10 samples : 10 trials, 50% success rate each)

→ from numpy import random
x = random.binomial (n=10, p=0.5, size=10)
print (x)
#output array ([5 6 4 5 7 3 4 6 5 5])

⑥ Visualization Example:-

```
from numpy import random
import matplotlib.pyplot as plt
import seaborn as sns
sns.displot(random.binomial(n=10, p=0.5, size=1000))
plt.show()
```



- plots a histogram (or density) of the binomial sample distribution.

Why Use It Instead of Normal Distribution?

- Binomial is discrete; good for counts of successes.
- Normal is continuous and may resemble binomial in shape when parameters are large.

Quick feature table!-

Distribution Type → Discrete (counts of successes)
parameters → n (trials), p (success probability)
output → Scalar or array (depending on size)
Use cases → Coin flips, quality checks, surveys
Visualization → Histogram via sns.displot or plt.hist

Numpy poisson Distribution:-

- poisson Distribution is a discrete probability distribution that models how many times an events occurs within a fixed interval (e.g. call per hour, arrivals per day) when events happen at a known constant average rate

Function Syntax:-

random.poisson (lam = , size =)

lam → expected rate (λ), the average number of events occurances in the interval.
size → Shape of the output (e.g. integer for 1D array, tuple for multidimensional)

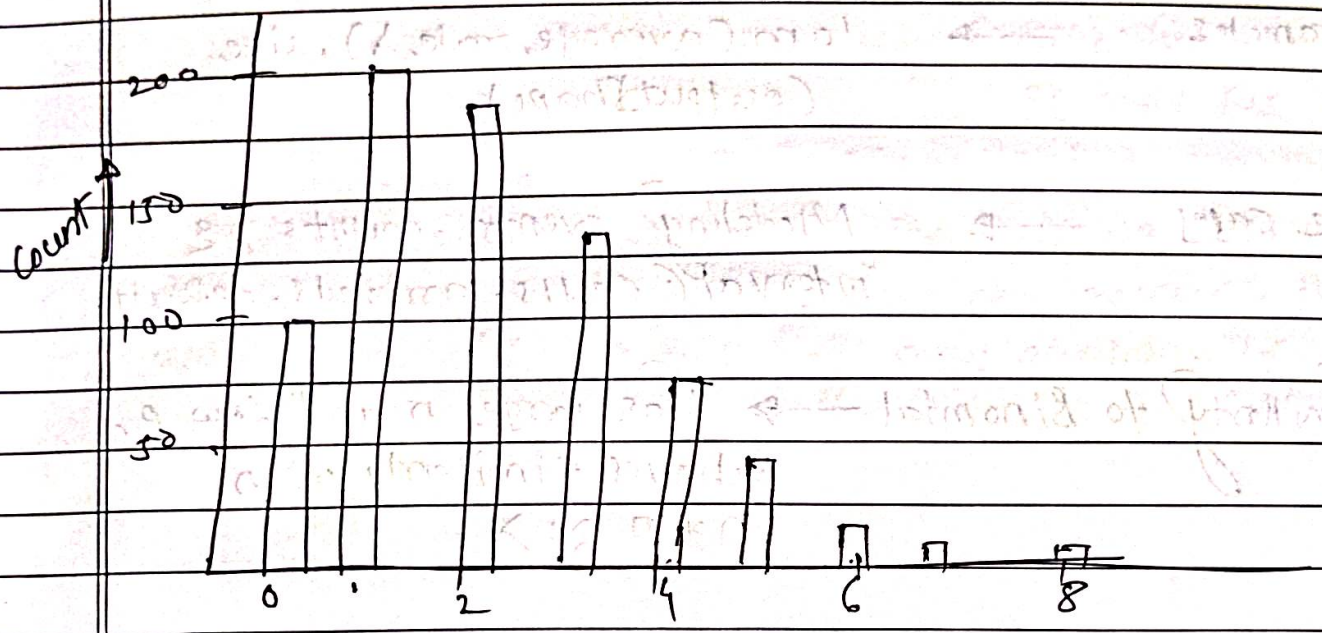
Example:- Q. 1D array of 10 samples, with an average of 2 events per interval.

→
```
from numpy import random.

X = random.poisson (lam=2, size=10)
print (X)
```

```
import matplotlib.pyplot as plt
import seaborn as sns

sns.displot(random.poisson(lam=2, size=1000))
plt.show()
```



- Generates a histogram-like plot showing the distribution of occurence.

Distribution Theory Insight:-

- Like the Binomial distribution with a very large number of trials (n) and very low success probability (p) the poisson distribution. approximates it when $n * p \approx \lambda$.

- Poisson is discrete, while Normal is continuous though for sufficiently large λ, the poisson distribution begins to resemble the normal distribution.

Quick Comparison Table:-

Feature	Details
Distribution Type	Discrete count of events in fixed intervals.
parameter	lam (average rate λ), size (output shape),
Use cases	Modeling events counts per interval (calls, arrivals, defects)
Similarity to Binomial	for large n and low p, behaves similarly when $n * p \approx \lambda$
Behavior vs. Normal	still discrete - but approximates Normal for large λ

Numpy Uniform Distribution:-

- The Uniform Distribution generate random numbers where every value in a specified range is equally likely.

- It's a continuous distribution, not discrete - you get floats evenly spread across the interval.

Function Syntax:-

```
random.uniform(low=0.0, high=1.0, size=None)
```

low — lower bound (inclusive) defaults to 0.0
High — Upper bound (exclusive) defaults to 1.0
size — Shape of output array (scalar for single value or tuple for arrays)

Samples are drawn from the half-open interval [low, high) — meaning low can appear, High usually not, though rounding may occasionally include it.

Example:-

```
from numpy import random.
```

Q. Single random float between 0.0 and 1.0

→

```
x = random.uniform()
print(x)
```

Q. 2x3 array of random floats between 0.0 & 1.0

→

```
arr = random.uniform(size=(2,3))
print(arr)
```

Q 1D array of random 5 numbers between 3 (inclusive) & 5 (exclusive)

→

```
arr = random.uniform(low=3, high=5, size=5)
print(arr)
```

Q. 2x3 array between numbers 10 (inclusive) & 20 (exclusive)

→ arr = random.uniform (low = 10, high = 20, size = (2,3))

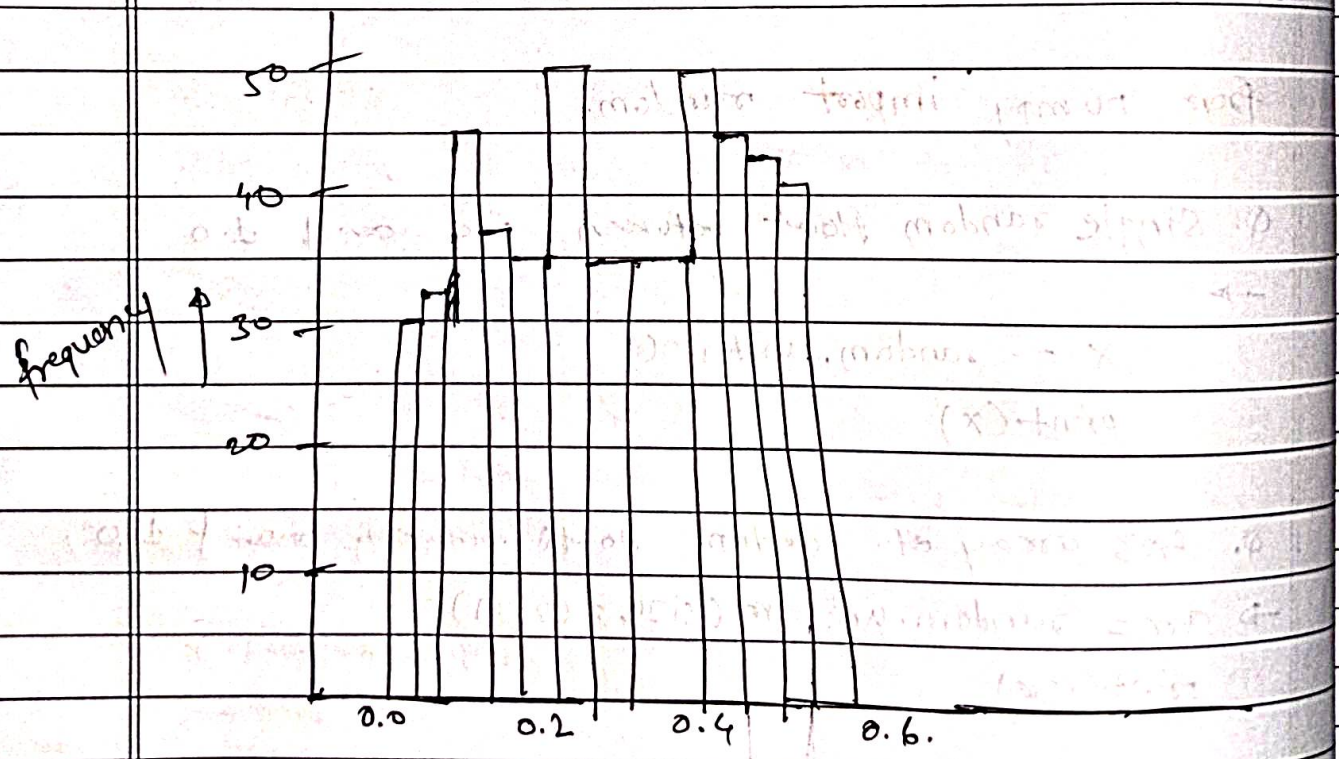
print(arr).

Visualizations :-

```
import matplotlib.pyplot as plt
import seaborn as sns.

data = random.uniform(size = 1000)
sns.histplot(data, bins = 30, kde = True)
plt.show()
```

Unifor Distribution [0,1] with KDE



Quick Comparision:-

Distribution Type ⟶ Continuous, uniform across [low, high]

parameter ⟶ low, high, size

Default ⟶ low = 0.0, high = 1.0, size = None

Range Behavior ⟶ inclusive of low, usually exclusive of High.

Use cases ⟶ Synthetic data, random intializations.

Summary:-

- Uniform distribution is perfect for scenarios where every numric outcome between two bounds should be equally probable.

- key parameters:- low, high & size to customize the range & shape of generated data.

- Visualization :- Histogram or KDE reveals a flat shape - confirming uniformity.

Logistic Distribution :-

- A continuous distribution similar to Normal but with heavier tails (higher chance of extreme values).

- Common in machine learning (e.g. logistic regression), growth models, and statistics.

Syntax

```
random.logistic(loc=0.0, scale=1.0, size=None)
```

loc → Mean (center of the distribution) → default 0
scale → spread / flatness (like standard deviation) → default 1
size → Shape of output array (e.g. (2,3))

Examples

Q1 single random value. (mean=0, scale=1)

```
from numpy import random.
print(random.logistic())
```

Q2. 2x3 array with mean = 1 & scale = 2

→

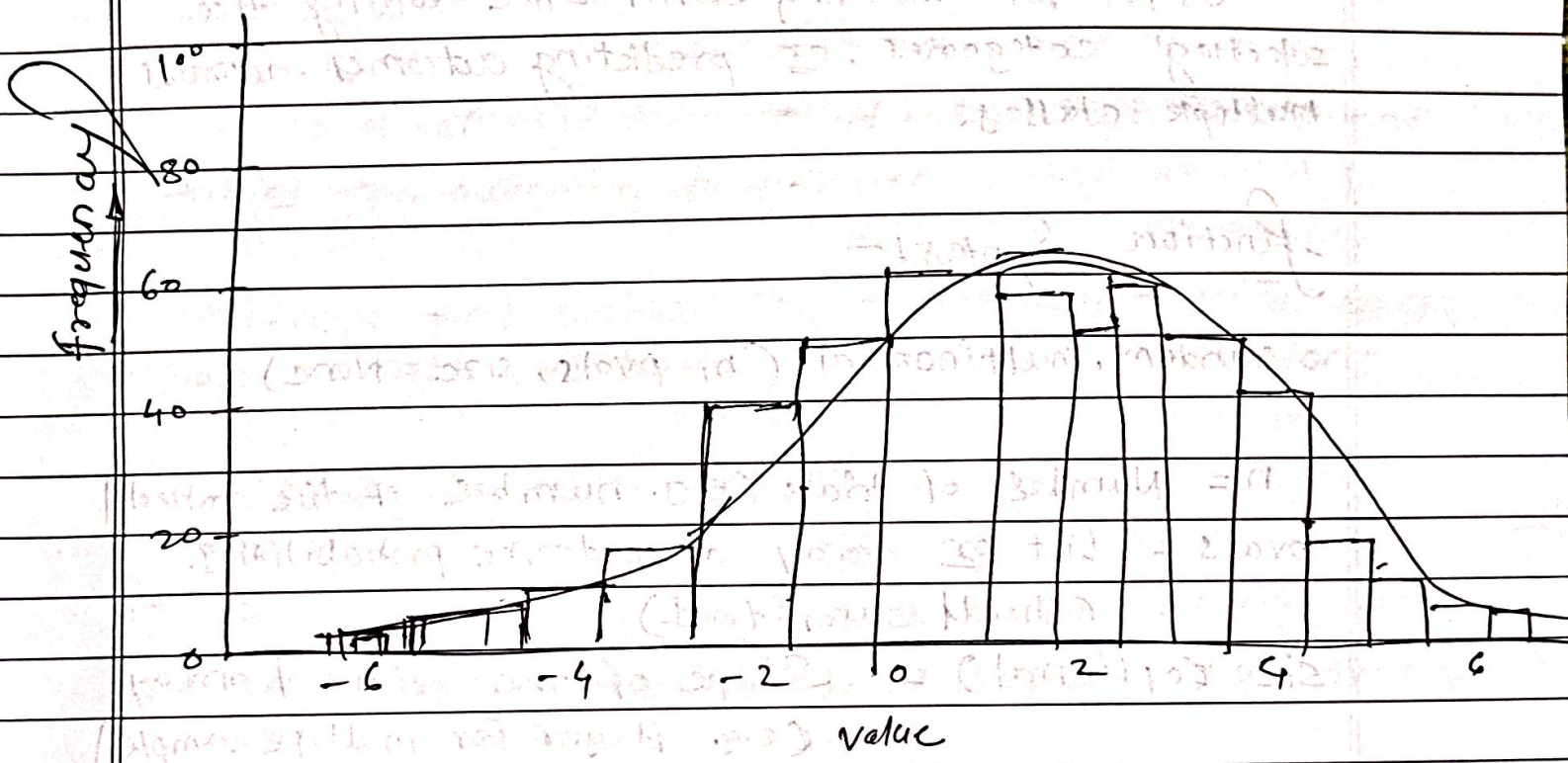
```
arr = random.logistic(loc=1, scale=2, size=(2,3))
print(arr)
```

Visualization!-

```
import matplotlib.pyplot as plt
import seaborn as sns
from numpy import random
```

```
# Generate 1000 samples
data = random.logistic (loc=0, scale=1, size=1000)

# plot distribution
sns.histplot (data, bins=30, kde=True)
plt.show()
```



#Logistic Vs Normal

- Both look bell-shaped
- logistic has ~~fatter~~ Fatter tails ⟶ gives more extreme values.
- Useful when you want slightly more variability than normal.

key point :-
- Continuous, bell-shaped distribution
- parameters: loc(mean), scale (spread)
- Heavier tails than Normal
- Used in ML (logistic regression) & growth models

Numpy Multinomial Distribution :-

- A generalization of the binomial distribution for scenarios with more than two possible outcomes.

- Useful for modeling events like rolling dice, selecting categories, or predicting outcomes, across multiple classes.

Function Syntax :-

random.multinomial (n, pvals, size = None)

n = Number of trials (e.g. number of dice rolled)
pvals = List or array of outcome probabilities. (should sum to 1)
size (optional) = Shape of the returned array (e.g. integer for multiple samples)

Example

Q. Simulate rolling a die 6 times.

→

```
from numpy import random
x = random.multinomial (n=6, pval = [1/6]*6)
print (x)
# output
[1, 0, 2, 1, 1, 1] — Counts per side of a die.
```

- you'll get one count per category - each index shows how many times that outcome occured.

Key Insight:-

- Unlike random.choice(), this returns an array of counts across categories.

- It effectively runs n id independent trials and tallies the outcome frequencies

- Shape and probabilities mirror what you'd except from multiple simultaneous binomial distributions.

Quick Comparison Table:-

Distribution Type	Discrete, multi-category counts.
parameters	n (trials), pvals (probabilities), size (optional
output	counts per category across n trials
Example Use Case	Rolling dice, multi-label outcomes, categorical sampling
Relation to Binomial	Reduce to binomial when there are only two categories.

Numpy Exponential Distribution :-

- A continuos probability distribution used to model the time until the next event, such as failure times, ~~until the next~~ waiting times between arrivals etc.

- Related to poisson distribution : while Poisson models counts, Exponential models the intervals between those events.

Function Syntax :-

random.exponential (scale = 1.0, size = None)

scale (β) = Defines the average interval (mean = β. it's the inverse of the rate parameter ($\lambda = 1/\beta$).

size = Output shape (None gives a single value else an array)

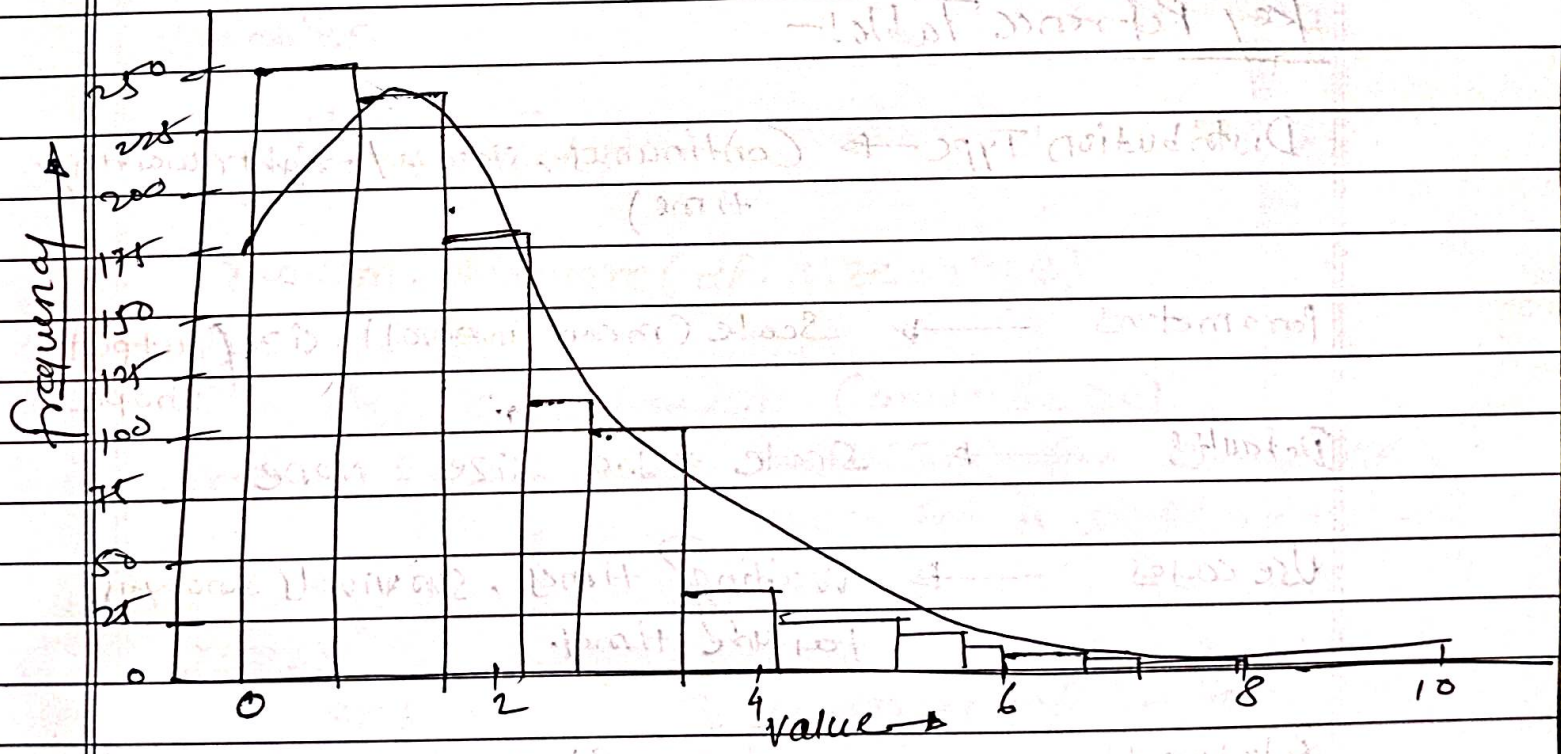
Example :-

```
from numpy import random
# 2x3 array, average interval (scale) = 2
x = random.exponential (scale=2, size=(2,3))
print (x)
```

⑥ Visualization:-

```
import matplotlib.pyplot as plt
import seaborn as sns
from numpy import random

data = random.exponential(scale=2, size=1000)
sns.histplot(data, bins=30, kde=True)
plt.show()
```



- This plot shows the typical skewed (right-tailed) shape of the exponential distribution, where most values cluster near zero and gradually taper off.

Distribution Relationship:-

poisson Vs Exponential:-

poisson : models how many events occur in a fixed time

Exponential : Models the waiting time between events.

Key Refrence Table:-

Distribution Type → Continuous, skewed right (waiting time)

Parameters → Scale (mean interval), size (output shape)

Defaults → Scale = 1.0, size = None

Use cases → Waiting times, survivals analysis, failure times.

Relationship → poisson (Counts) ↔ Exponential (intervals)

Summary:-

- Use random.exponential() when you need to simulate time intervals between random events.

- Rember! larger scale = longer expected intervals.

- The distribution is right-skewed, with a high occurrence near zero & a long tail.

Numpy Chi-Square Distribution:-

- A continuous probability distribution often used in statistics - particularly in hypothesis testing (like goodness-of-fit or independence tests) to evaluate observed vs. expected data. It's derived by summing the square of independent standard normal variables.

-function Syntax:-

random.chisquare(df, size=None)

df = Degree of freedom (must be >0)
size = Output shape (e.g. (2,3) for arrays or None for a single value)

Example:-

Q1. 2x3 array with 2 degrees of freedom.

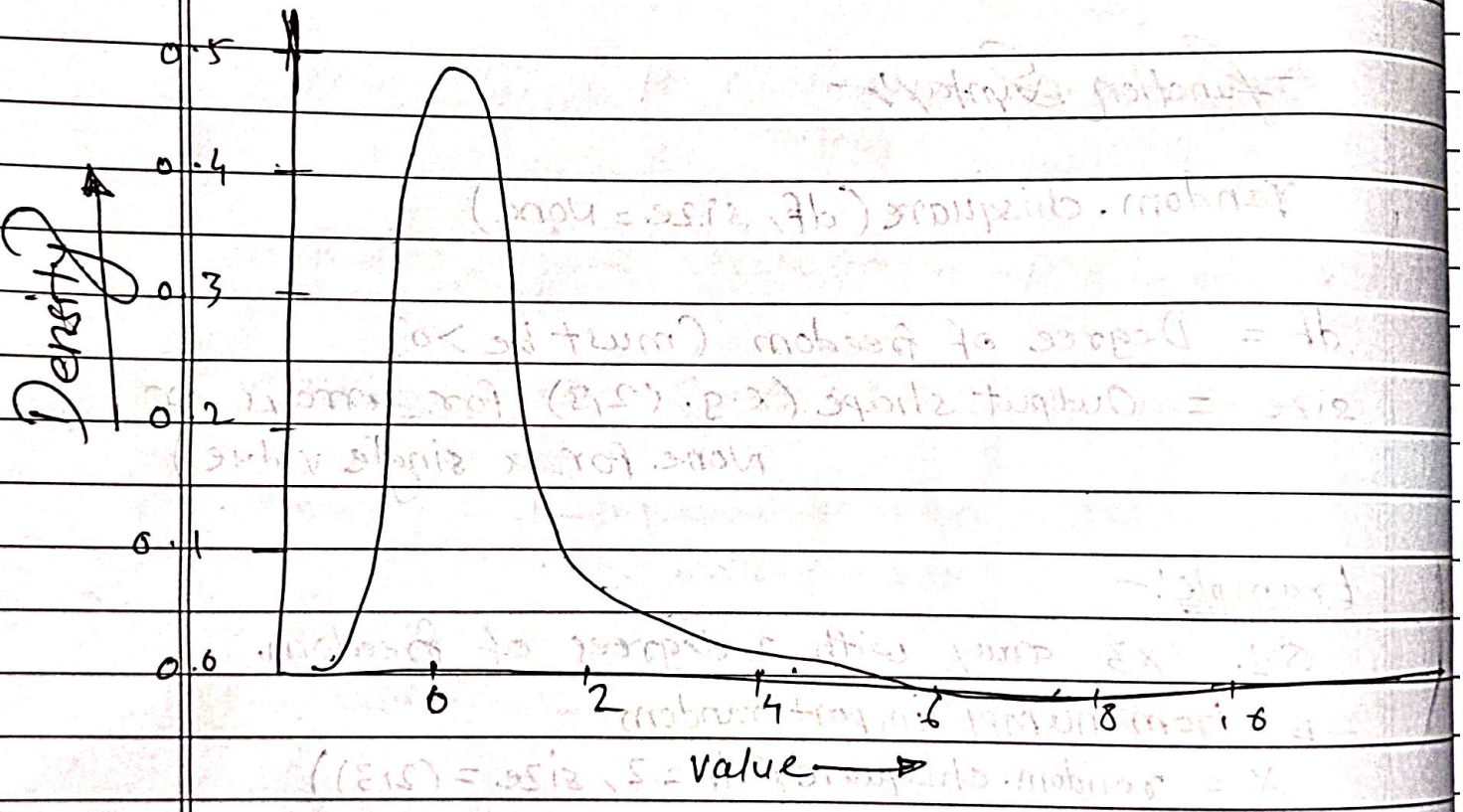
→
```
from numpy import random
X = random.chisquare(df=2, size=(2,3))
print(X)
```

Generate an array of chi-square distributed values with df=2.

Visualization Example:-

```
import matplotlib.pyplot as plt
import seaborn as sns
from numpy import random.

# Gererate 1000 samples with df=1
data = random.chisquare(df=1, size=1000)
# plot KDE
sns.displot(data, kind="kde")
plt.show()
```



- Produces a smooth right-skewed curve, typical for chi-square distributions with low degrees of freedom.

Theory & Behaviors:-

- The chi-square distribution is skewed to the right, particularly when df is low, as df increases the distribution becomes more symmetric.

mean = df

variance = 2 * df

Feature tables:-

Distribution Type	⟶	Continuous, skewed-right.
Parameters	⟶	df (degree of freedom), size
Default Behavior	⟶	scalar output if size = None.
Use Cases	⟶	Hypothesis tests, variance testing, contingency tables.
Characteristics	⟶	Skewed right; more symmetric as df grows.

Numpy Rayleigh Distribution:-

- A continuous, right-skewed ideal for modeling the magnitude of a 2D vector whose components are independent, zero-mean normals — commonly used in signal processing and wind speed modeling.

Function Syntax:-

random.rayleigh(scale=1.0, size=None)

scale = Standard deviation or mode (must be non-negative, default=1)

size = Shape of the result (scalar if None, else array of given dimension)

Example

Q. Generate a 2x3 array of raleigh-distributed values (scale=2)

→

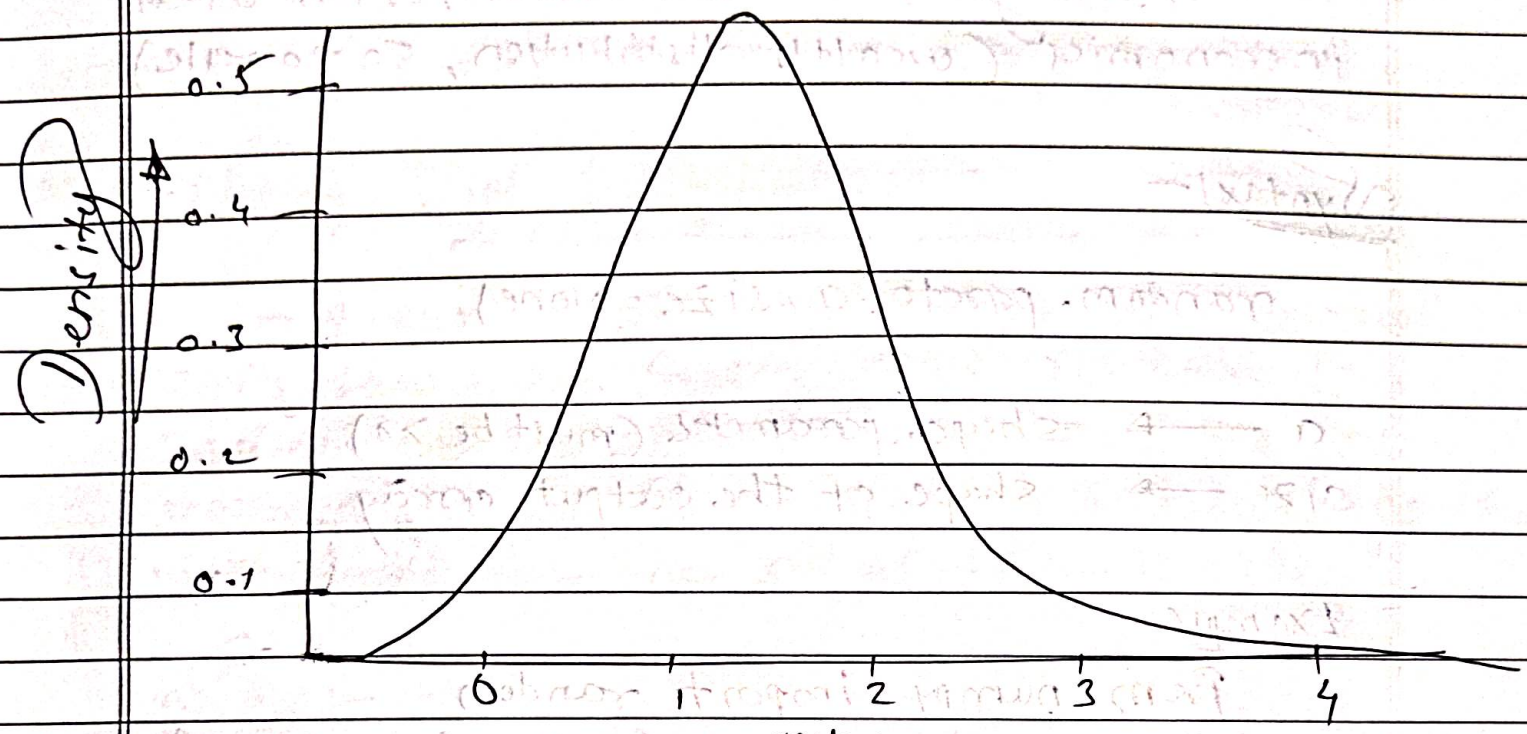
```
x = random.rayleigh(scale=2, size=(2,3))
print(x)
```

Visualization:-

```
import matplotlib.pyplot as plt
import seaborn as sns
from numpy import random.

# Visualization example: kDEplot
```

```
sns.displot(random.rayleigh(size=1000), kind = 'kde')
plt.show().
```



Summary Table:-

Type	⟶ Continuous, right-skewed distribution.
parameters	⟶ scale (spreed / mode), size (output shape
Visualization	⟶ Histogram / KDE shows peak near low values with long tail.
Use Cases	⟶ Signal amplitude, wind speed, physical magnitude modeling?
Relation to Others	⟶ Equivalent to χ^2 distribution (df = 2) when scale = 1

Numpy Pareto Distribution:-

- A heavy-tailed distribution, often used in economics (wealth distribution, 80-20 rule)

Syntax:-

random.pareto(a, size=None)

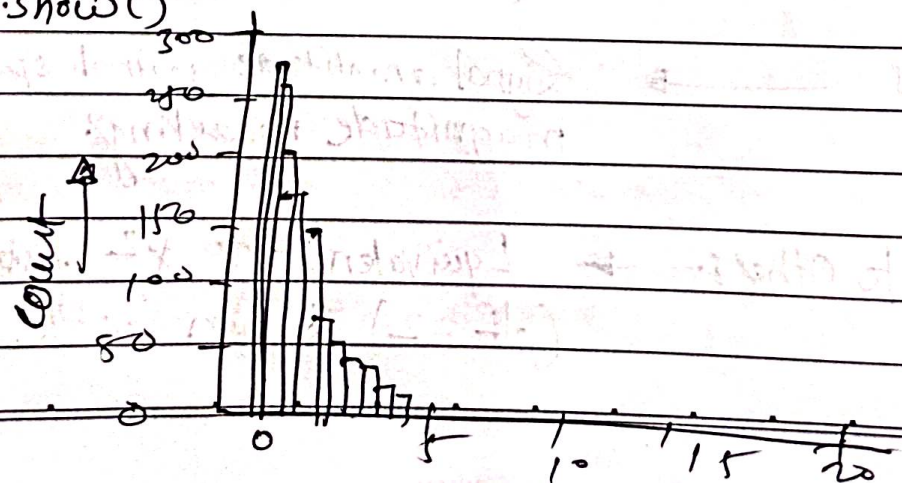
a ⟶ shape parameter (must be >0)
size ⟶ shape of the output array

Example

```
from numpy import random
x = random.pareto(a=2, size=(2,3))
print(x)
```

Visualization:-

```
from numpy import random
import matplotlib.pyplot as plt
import seaborn as sns
sns.displot(random.pareto(a=2, size=1000))
plt.show()
```



Behaviour :-

- Many small values near 0.
- Few very large values (long right tail)

Numpy Zipf Distribution :-

- A discrete probability distribution following Zipf's Law - the frequency of an item is inversly proportional to its rank (i.e. the 2nd most common occurs at 1/2 the frequency of the most common, the 3rd at 1/3, etc)

Sytax :-

random.zipf(a, size=None)

a ⟶ Distribution parameter (must be >1) controls the skew.

size ⟶ Shape of the returned array.

Example :-

Q. Create a 2x3 array with zipf distribution (a=2)

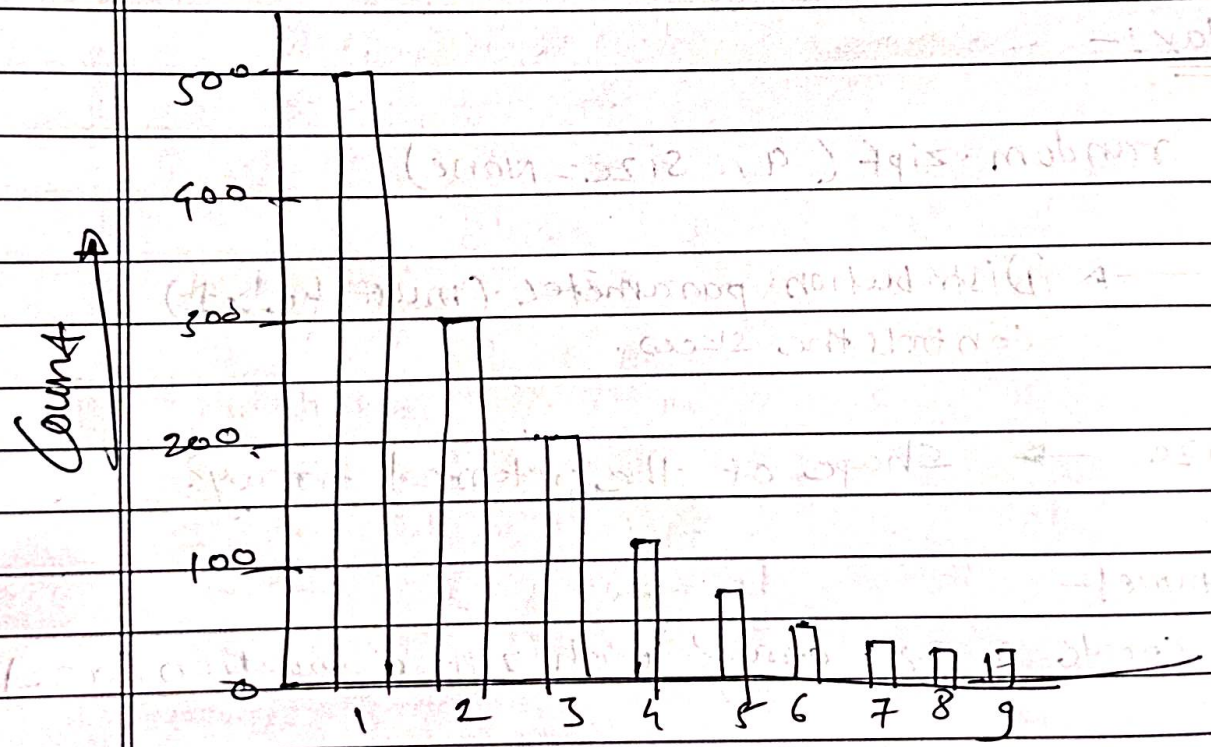
⟶

```
from numpy import random
x = random.zipf(a=2, size(2,3))
print(x)
```

Visualizations—

Sample 1000 points but plotting only ones with value < 10 for more meaningful chart

→
```
from numpy import random
import matplotlib.pyplot as plt
import seaborn as sns
x = random.zipf(a=2, size=1000)
sns.distplot(x[x<10])
plt.show()
```



Numpy ufuns :- (universal functions)

- ufuncs ("universal functions") are Numpy function that apply element-wise operations directly to ndarray objects.

- They enable vectorization, which means applying operations across entire arrays at once – this is much faster than looping element by element in python.

- ufuncs also support broadcasting and methods like reduce, accumulate, and optional parameter such as where, dtype, and out for flexible execution.

Why Are ufuncs Helpful?

Speed - They leverage low-level optimizations for speed calculations, avoiding slow python loops.

Convenience - You can perform operation on arrays of different shapes (thank to broadcasting) and apply ufuncs conditionally (using where) or with controlled output types and destination (via dtype and out)

Key Example : Vectorized Addition :-

- Without ufuncs, you'd use something like:

```
z = [ ]
for i,j in zip(x,y):
    z.append(i+j)
```

- With a ufunc:

```
z = np.add(x,y)
```

This operates on entire arrays in one statement. cleaner, faster, and more expressive.

In essence

- Ufuncs are powerful, fast, and expressive. ideal for efficient array computations in Numpy.

Creating your own function in (Ufuncs) Numpy :-

- You can convert any normal python function into a Numpy universal function (ufunc) using np.frompyfunc()

- The frompyfunc() method requires three parameters:

1) function - the python function you define.
2) Inputs — Number of input arguments (array
3) outputs - Number of output arrays.

Example:-

```
import numpy as np
def myadd(x,y):
    return x+y
myadd = np.frompyfunc(myadd, 2, 1)
print(myadd([1,2,3], [4,5,6]))
```

- This applies myadd element-wise across arrays, just like numpy's built-in ufuncs.

- To confirm whether a function is a ufunc, you can check its type:

```
import numpy as np
print(type(np.add))          # <class 'numpy.ufunc'>
print(type(np.concatenate)) # not a ufunc.
```

In code, you can even write

```
If type(np.add) == np.ufunc:
    print('add is ufunc')
else:
    print('add is not ufunc')
```

In essence,

- np.frompyfunc() allows you to wrap your own Python function into a fast, element-wise ufunc - with customizable numbers of inputs & outputs. And you can easily test if a given function is a genuine

Numpy ufunc by cheeking its type.

Numpy ufuncs:-

Addition (np.add):-

```
arr1 = np.array ([10,11,12,13,14,15])
arr2 = np.array ([20,21,22,23,24,25])
newarr = np.add (arr1, arr2)
print (newarr)
    → # [30,32,34,36,38,40]
```

Subtraction (np.substract):-

```
arr1 = np.array ([10,20,30,40,50,60])
arr2 = np.array ([20,21,2423,25])
print (newarr)
    → # [-10,-1,8,17,26,35]
```

Multiplication (np.multiply)

```
arr1 = np.array ([10,20,30,40,50,60])
arr2 = np.array ([20,21,22,23,24,25])
new arr = np.multiply (arr1, arr2)
print (new arr)
    → # [200,420,660,920,1200,1500]
```

Division (np.devide):-

```
arr1 = np.array ([10,20,30,40,50,60])
arr2 = np.array ([3,5,10,8,2,33])
newarr = np.divide (arr1, arr2)
print (newarr)
    → # [3.33,4,3,5,1.81]
```

Power (np.power) :-

```
arr1 = np.array([10,20,30,40,50,60])
arr2 = np.array([3,5,6,8,2,33])
newarr = np.power(arr1, arr2)
print(newarr)
```
→ # [1000, 3200000, 729000000, 655360000000, 2500, 0]

Remiander (Modulus: np.mod or np.remainder) :-

```
arr1 = np.array([10,20,30,40,50,60])
arr2 = np.array([3,7,9,8,2,33])
Newarr = np.mod(arr1, arr2)
print(newarr)    # [1, 6, 3, 0, 0, 27]
# same result with np.remainder()
```

Quotient & Remainder (np.divmod)

```
arr1 = np.array([10,20,30,40,50,60])
arr2 = np.array([3,7,9,8,2,33])
newarr = np.divmod(arr1, arr2)
print(newarr) # (array([3,2,3,5,25,1]), array([1,6,3,0,0,27]))
```

Absolute Value (np.absolute or np.abs) :-

```
arr1 = np.array([-1, -2, 1, 2, 3, -4])
newarr = np.absolute(arr)
print(newarr)    # [1, 2, 1, 2, 3, 4]
```

Quick Takeaways :-

	Feature	What It Does	Example outcome
1)	Vectorized	operates on entire arrays in one go	fast and concise array operation.
2)	Element-wise	Applies function to each element	Like doing +, −, * etc on list in a loop.
3)	Conditional	Apply operations selectively on elements.	—

- These ufuncs provide a functional & efficient way to apply common arithmetic operation across whole arrays — no loops required, just clean & expressive code.

Rounding Decimals (with examples) :-

- Numpy offers five primary ufuncs to round or adjust decimal values.

np.trunc() & np.fix() :-

- Remove decimal parts by rounding towards zero (i.e. toward 0)

Example :-

```
np.trunc([-3.1666, 3.6667])
    # [-3, 3]   # same result with fix().
```

④ np.around() :-

- Round to the nearest value, rounds up if the discarded digit is ≥5

e.g.

```
np.around(3.1666, 2)
    # [3.17, 2]
```

⑤ np.floor() :-

- Round decimal values down to the next lower integer. Always round down (toward $-\infty$)

e.g.

```
np.floor([-3.1666, 3.6667])
    # [-4, 3]
```

⑥ np.ceil() :-

- Round decimal values up to the next higher integer. Always round up (toward $+\infty$)

e.g.

```
np.ceil([-3.1666, 3.6667])
    # [-3, 4]
```

Numpy Logs:-

- np.log2(arr):-

- Computes the logarithm base 2 for each element.

e.g.

```
arr = np.arrange(1,10)
print(np.log2(arr))
```

This returns the base-2 logarithms of the numbers 1 through 9.

- np.log10(arr):-

- Computes the logarithm base 10 for each element.

e.g.

```
arr= np.arrange(1,10)
print(np.log10(arr))
```

- This returns the base-10 logarithms of the numbers 1 through 9

- np.log(arr):-

- Computes the natural logarithm (base e) for each element

e.g.

```
arr = np.arrange(1,10)
print(np.log(arr))
```

- This returns the natural logarithms of the numbers 1 through 9.

Custom base logarithm:-

- Numpy doesn't offer a direct ufunc for arbitary base logs, but you can create your own using np.frompyfunc().

e.g.

```
from math import log
import numpy as np
nplog = np.frompyfunc(log,2,1)
print(nplog(100,15))
```

- This applies Python's math.log() to compute logarithms with any base (input number and base), producing $\log_{15}(100)$ in this case.

At a Glance:-

function	what it does.
np.log2()	→ Computes log base 2 element-wise
np.log10()	→ Compute log base 10 element-wise
np.log()	→ Computes natural log (base e)
frompyfunc()	→ Enables custom-base log functions.

Summations :-

Summation Vs Addition :-

- Addition (np.add) operates between corresponding elements of two ~~yeat~~ arrays.

e.g.

```
arr1 = np.array([1,2,3])  ——*
arr2 = np.array([1,2,3])  ——*
np.add(arr1, arr2)
# → [2,4,6]
```

- Summation (np.sum) aggregates values across one or more arrays.

e.g.

```
np.sum(arr1, arr2)
# → 12
```

Summation Over an Axis :-

Summing across rows or columns in a multidimensional array.

e.g.

```
np.sum([arr1, arr2], axis=1)
# → [6,6]
```

Cumulative Sum (np.cumsum) :-

- Builds a running total over array elements

e.g.

```
arr = np.array ([1,2,3])
np.cumsum (arr)
# → [1,3,6]
```

In Brief :-

Ufunc	What It Does	Example output
np.add()	Element-Wise addition of two arrays	[2,4,6]
np.sum()	Total sum across arrays or along specified axis	12 or [6,6]
~~np.cusum~~		
np.cumsum	produces cumulative totals along an array	[1,3,6]

- These function let you effortlessly perform both aggregate and element-wise summations in a clean efficient way - no loops required.

Products :-

np.prod() :-

computes the product of all elements in an array.

e.g.

```
arr = np.array([1,2,3,4])
x = np.prod(arr)
print(x)  # → 24
```

product across multiple arrays :-

- You can pass a list of arrays to calculate the combined product.

e.g.

```
arr1 = np.array([1,2,3,4])
arr2 = np.array([5,6,7,8])
x = np.prod([arr1, arr2])
print(x)  # → 40320
```

product along an axis :-

- By specifying axis = 1, you get the product for each sub-array.

e.g.

```
arr1 = np.array([1,2,3,4])
arr2 = np.array([5,6,7,8])
newarr = np.prod([arr1, arr2], axis=1)
print(newarr)
→ # [24, 1680]
```

Cumulative product (np.cumprod()):-

- Generates a running product (e.g. partial products).

e.g.

```
arr = np.array([5,6,7,8])
newarr = np.cumprod(arr)
print(newarr) → # [5, 30, 210, 1680]
```

Quick Overview:-

np.prod() → multiplies all elements in an array.

np.prod([...]) → Multiplies elements across multiple ~~arrays~~ ~~axis~~. arrays.

axis parameter → Applies product per sub-array along axis.

np.cumprod → Returns cumulative products in sequence.

In short:-

- These ufuncs give you powerful, loop-free ways to compute total, cumulative products — either across arrays or along specified axes.

1 Differences :-

- A discrete difference substracts each element from the next in an array - effectively capturing successive changes.

```
import numpy as np
arr = np.array([10,15,25,5])
newarr = np.diff(arr)
print(newarr)
#--> [5,10,-20]
```

Explaination :- 15-10=5, 25-15=10, 5-25=-20

- You can apply the operation multiple times by setting the parameter n.

```
arr = np.array([10,15,25,5])
newarr = np.diff(arr, n=2)
print(newarr)
#--> [5, -30]
```

Explaination, 1st pass gives [5,10,-20]
2nd pass (n=2), 10-5=5, -20-10=-30,

@ In plain Terms!-

np.diff() computes the differences between consecutive elements in an array.

- The n parameter determines how many times the differential operation repeats. (i.e. first-order, second-order etc)

- This operation is handy to detect trends or changes in data sequences - like how values changes over time, successive increases/decreases, or assessment of higher-order differences.

Numpy Ufunc - Lowest Common Multiple (LCM):-

@ Basic LCM of two numbers:-

```
import numpy asnp
num1 = 4
num2 = 6
x = np.lcm(num1, num2)
print(x)
```

⟶ # 12 is the smallest number divisble by both 4 & 6

@ LCM accross an entire array using .reduce():
You can compute the LCM of all elements in an array.

```
arr = np.array([3,6,9])
x = np.lcm.reduce(arr)
print(x)  ⟶ # 18
```

Here, the LCM of 3, 6, and 9 is 18

⊙ LCM over a range of values:-

- for example, to find the LCM of integers 1 trough 10.

```
arr = np.arange(1,11)
x = np.lcm.reduce(arr)
print(x)
```

⟶ # returns LCM of all number from 1 to 10

- This aggregates the LCM across the full range.

function

np.lcm(a,b) → finds the LCM of two numbers	→ np.lcm(4,6) → 12	
np.lcm.reduce(arr) → computes the LCM across all elements in arr	→ [3,6,9] → 18	
np.lcm.reduce (range(1,11)) → Gets LCM for a sequence 1 to 10	→ (LCM of 1-10)	

Numpy ufunc - Greatest Common Divisor (GCD):-

Basic GCD of Two numbers

```
import numpy as np
num1 = 6
num2 = 9
x = np.gcd(num1, num2)
print(x)
```

→ # 3 The GCD of 6 & 9 is 3 because it's the largest number that devides both evenly.

GCD over an Array using .reduce()

```
arr = np.array([20, 8, 32, 36, 16])
x = np.gcd.reduce(arr)
print(x)
```

→ # 4 is the greatest common divisor of all elements in the array.

In short:-

np.gcd(a, b) → computes the GCD of two integers (or element-wise GCD across arrays)

np.gcd.reduce(array) → computes the GCD of all elements in the array.

Trigonometric functions :-

Sin (np.sin) :-

- computes the sine of each element (in radians)

```
x = np.sin(np.pi/2)
```

→ # → 1

Applied to Arrays :-

```
arr = np.array([np.pi/2, np.pi/3, np.pi/4, np.pi/5])
x = np.sin(arr)
```

→ # [1, 0.866, 0.707, 0.587]

Converts Degree ↔ Radians :-

- Numpy expects input in radians, but offers convenient conversion functions.

- np.deg2rad(), converts degrees to radians.

```
arr = np.array([90, 180, 270, 360])
print(np.deg2rad(arr))
```

→ # [π/2, π, 3π/2, 2π]

- np.rad2deg(), converts radians to degree.

```
arr = np.array([np.pi/2, np.pi, 1.5*np.pi, 2*np.pi])
print(np.rad2deg(arr))
```

→ [90, 180, 270, 360]

⑥ Inverse Trigonometric function (arcsin, arccos, arctan):-

- Compute the angle (in radians) given a trigonometric value.

- Example using arcsin.

```
x = np.arcsin(1)    # π/2
```

- Applied across arrays.

```
arr = np.array([1, -1, 0.1])
print(np.arcsin(arr))  # [1.57, -1.57, 0.1]
```

⑦ Hypotenuse (np.hypot()):-

- Calculate the hypotenuse from the lengths of the two perpendicular sides.

```
base = 3
prep = 4
print(np.hypot(base, prep)   # 5.0
```

- These ufuncs offer fast, element-wise trigonometric and geometric transformations — ideal for clean, vectorized operations without loops.

Quick overview:-

np.sin() ⟶ Element-wise sine of angles (radi

np.deg2rad/ np.rad2deg ⟶ converts between degrees and radians

np.arcsin()/ np.arccos()/ np.arctan() ⟶ compute angles from sin, cos, tan, values (in radian)

np.hypot() ⟶ compute hypotenuse using pythagorean theorem.

Hyperbolic functions:-

1. Hyperbolic Sine (np.sinh):-

- Calculate the hyperbolic sine for each element (in radians)

e.g.

```
x = np.sinh(np.pi/2)
```

2. Hyperbolic Cosine (np.cosh):-

- Compute the hyperbolic cosine for each element

e.g.

```
arr = np.array([np.pi/2, np.pi/3, np.pi/4, np.pi/5])
x = np.cosh(arr)
```

Hyperbolic Tangent (np.tanh):-

- Calculate the hyperbolic tangent of each value Example Similarly follows. the same style of sinh, cosh on an array.

Inverse Hyperbolic function:-

Inverse Hyperbolic Sine (np.arcsinh):-

- Returns the value whose hyperbolic sine equals the input.

```
X = np.arcsinh(1)
```

Inverse Hyperbolic Cosine (np.arccosh) & Inverse Hyperbolic Tangent (np.arctanh)

- Apply these element-wise on arrays to compute corresponding inverse hyperbolic functions.

```
arr = np.array([0.1, 0.2, 0.5])
X = np.arctanh(arr)
```

- outputs the radians values whose hyperbolic tangent matches each array entry.

Inshort:-

np.sinh, np.cosh(), np.tanh()	→ compute hyperbolic sine, cosine & tangent, elementwise.
np.arcsinh(), np.arccosh(), np.arctanh(),	→ compute inverse hyperbolic functions (radians. results)

Numpy Set Operations :-

- Numpy provides functions to perform set operations on arrays, such as union, intersection, and difference. These operations are useful for handling unique elements and performing mathematical set operations efficiently.

① Creating Sets in Numpy :-

- To create a set from an array, use the np.unique() function, which returns the sorted unique elements of an array.()

e.g.

```
arr = np.array([1, 2, 2, 3, 4, 4, 5])
uni-ele = np.unique(arr) # [1, 2, 3, 4, 5, 6]
```

② Set operations :-

(1) Union (np.union1d()) :-

- Returns the sorted union of two arrays i.e. all unique elements present in either array.

e.g.

```
arr1 = np.array([1, 2, 3])
arr2 = np.array([3, 4, 5])
union = np.union1d(arr1, arr2)
    # [1, 2, 3, 4, 5, 6]
```

(2) Intersection (np.intersect1d()) :-

- Returns the sorted, unique values that are present in both input arrays. ().

e.g.

```
intersection = np.intersect1d(arr1, arr2)
arr1 = np.array([1,2,3])
arr2 = np.array([3,4,5])
# 3
```

(3) Difference (np.setdiff1d()) :-

- Returns the ~~record~~ sorted, unique values in the first array that are not in the second array.

e.g.

```
arr1 = np.array([1,2,3])
arr2 = np.array([3,4,5])
diff = np.setdiff1d(arr1, arr2)
# [1,2]
```

(4) Symmetric Difference (np.setxor1d()) :-

- Return the Sorted, unique values that are in only one of the input arrays, but not both ().

```
arr1 = np.array([1,2,3])
arr2 = np.array([3,4,5])
sym_diff = np.setxor1d(arr1, arr2)
# [1,2,4,5]
```

@ In Short!-

np.union1d() ⟶ Union of two arrays (all unique elements)

np.intersect1d() ⟶ Intersection of two arrays (common elements)

np.setdiff1d() ⟶ Element in the first array but not the second

np.setxor1d() ⟶ Element in either array but Not ~~in the second~~ Both.

- These function allowed for efficient set operations on arrays, enabling quick computations without the need for explict loops.