**Numpy -Python**

**Introduction to NumPy**

* **NumPy** (Numerical Python) is a library for the Python programming language that provides support for large, multi-dimensional arrays and matrices, along with a collection of mathematical functions to operate on these arrays.
* It is a fundamental package for scientific computing in Python.
* NumPy was created in 2005 by Travis Oliphant. It is an open-source project and you can use it freely.
* NumPy is a Python library used for working with arrays.
* NumPy is a Python library and is written partially in Python, but most of the parts that require fast computation are written in C or C++.
* The source code for NumPy is located at this github repository <https://github.com/numpy/numpy>

### Key Features of NumPy

* **N-dimensional array objects (ndarray).**
* **Mathematical functions: Operations on arrays.**
* **Linear algebra, Fourier transform, and random number capabilities.**

## **What is an Array in Python?**

* An array is a collection of items stored at contiguous memory locations. The idea is to store multiple items of the same type together.

## **Why Use NumPy?**

* In Python we have lists that serve the purpose of arrays, but they are slow to process.
* NumPy aims to provide an array object that is up to 50x faster than traditional Python lists.
* The array object in NumPy is called ndarray, it provides a lot of supporting functions that make working with ndarray very easy.
* Arrays are very frequently used in data science, where speed and resources are very important.

## **Why is NumPy Faster Than Lists?**

* NumPy arrays are stored at one continuous place in memory unlike lists, so processes can access and manipulate them very efficiently.
* This behaviour is called locality of reference in computer science.
* This is the main reason why NumPy is faster than lists. Also, it is optimized to work with latest CPU architectures.
* No need for data type checking.
* Item wise computations available.

**How NumPy Relates to Other Fields**

#### Data Science

* NumPy is foundational for data science as it provides efficient storage and manipulation of numerical data.
* It's used in data preprocessing, statistical analysis, and implementing algorithms that require fast numerical computations.
* Libraries like pandas and scikit-learn are built on top of NumPy, leveraging its capabilities to handle large datasets and perform complex operations efficiently.

#### Scientific Computing

* NumPy's support for multi-dimensional arrays and a rich set of mathematical functions makes it ideal for scientific computing.
* Researchers and engineers use NumPy for simulations, modeling, and solving mathematical problems.
* It integrates well with other scientific libraries like SciPy, which builds on NumPy to provide additional functionality for optimization, integration, and statistics.

### Installation

To install NumPy, you can use pip:

pip install numpy

### Importing NumPy

* To start using NumPy, you need to import it:

import numpy as np

* Now the NumPy package can be referred to as np instead of numpy.

## **Create a NumPy ndarray Object**

* NumPy is used to work with arrays. The array object in NumPy is called ndarray.
* We can create a NumPy ndarray object by using the array() function.
* An array that has 0-D arrays as its elements is called uni-dimensional or 1-D array.
* These are the most common and basic arrays.
* # 1D array

arr\_1d = np.array([1, 2, 3, 4, 5])

print(arr\_1d)

print(type(arr\_1d))

**type():** This built-in Python function tells us the type of the object passed to it. Like in above code it shows that arr is numpy. ndarray type.

## **0-D Arrays**

* 0-D arrays, or Scalars, are the elements in an array. Each value in an array is a 0-D array.
* import numpy as np  
    
  arr = np. array(42)  
    
  print(arr)

**2D array**

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

arr\_2d = np.array([[1, 2, 3], [4, 5, 6]])

print(arr\_2d)

## **3-D arrays**

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

import numpy as np  
arr = np.array([

[[1, 2, 3], [4, 5, 6]],

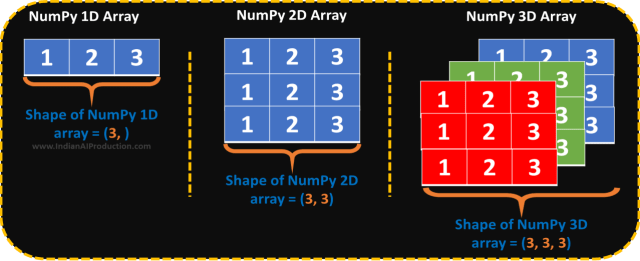


[[1, 2, 3], [4, 5, 6]]

])  
print(arr)

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

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



**Ragged Array**

* If you have a list of lists (a 2D array) where the inner lists have different numbers of elements, it is considered a "ragged" or "jagged" array.
* NumPy requires arrays to be rectangular (i.e., each row must have the same number of columns).
* To handle ragged arrays, you can use Python's built-in lists or, for certain operations, the np.object dtype.

Eg:

ragged\_array\_np = np. array([ [1, 2, 3], [4, 5], [6, 7, 8, 9] ], dtype=object)

# Accessing elements

print(ragged\_array\_np[0]) # Output: [1, 2, 3]

print(ragged\_array\_np[1][1]) # Output: 5

print(ragged\_array\_np[2][:2]) #Output:[6,7]

**Properties of ndarray**

* **ndim**: Number of dimensions (axes) of the array.
* **shape**: A tuple of integers indicating the size of the array in each dimension.
* **size**: Total number of elements in the array.
* **dtype**: Data type of the array elements.

## **Check Number of Dimensions?**

* NumPy Arrays provides the ndim attribute that returns an integer that tells us how many dimensions the array 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]])  
d = np.array([[[1, 2, 3], [4, 5, 6]], [[1, 2, 3], [4, 5, 6]]])  
  
print(a.ndim)  
print(b.ndim)  
print(c.ndim)  
print(d.ndim)

## **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 arrays start with 0, meaning that the first element has index 0, and the second has index 1 etc.

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

## **Access 2-D Arrays**

* To access elements from 2-D arrays we can use comma separated integers representing the dimension and the index of the element.
* Think of 2-D arrays like a table with rows and columns, where the dimension represents the row and the index represents the column.

import numpy as np  
arr = np. array ([[1,2,3,4,5], [6,7,8,9,10]])  
print ('2nd element on 1st row: ', arr [0, 1])

## **Access 3-D Arrays**

* To access elements from 3-D arrays we can use comma separated integers representing the dimensions and the index of the element.

import numpy as np

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

**work:**

Get third and fourth elements from the following array and add them.

## **Negative Indexing**

Use negative indexing to access an array from the end.

Print the last element from the 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])

## **Slicing arrays**

* Slicing in python means taking elements 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, it considered 1
* Eg: arr = np.array([1, 2, 3, 4, 5, 6, 7])  
   print (arr[1:5])

## **Negative Slicing**

* Use the minus operator to refer to an index from the end:

Eg:

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

* # Slicing 2-D Arrays

arr = np.array([[1, 2, 3, 4, 5], [6, 7, 8, 9, 10]])

print (arr [1, 1:4])

Eg: arr = np.array([[1, 2, 3, 4, 5], [6, 7, 8, 9, 10]])  
 print(arr[0:2, 1:4])

* # Slicing 3-D Arrays

arr3=np.array([ [[1,2,3,4],[3,4,5,6]],

[[2,3,4,5],[6,7,8,9]] ])

[0:2,0,1:]

## **STEP**

* Use the step value to determine the step of the slicing:

Eg:

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

## **Data Types in Python**

By default Python have these data types:

* strings - used to represent text data, the text is given under quote marks. Eg: "ABCD"
* integer - used to represent integer numbers. Eg: -1, -2, -3
* float - used to represent real numbers. Eg: 1.2, 42.42
* boolean - used to represent True or False.
* complex - used to represent complex numbers. Eg: 1.0 + 2.0j, 1.5 + 2.5j

## **Data Types in NumPy**

NumPy has some extra data types, and refer to data types with one character, like i for integers, u for unsigned integers etc.

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

* i - integer
* b - boolean
* u - unsigned integer (i.e., zero and positive integers).astype(np.uint32)
* f - float
* c - complex float(Real+ imaginary part)
* m - timedelta

( The timedelta class in Python is part of the datetime module and represents the difference between two dates, times, or datetime objects. It is useful for performing arithmetic with date and time objects, such as adding or subtracting time intervals.)

* M - datetime
* O - object- (object data type is a general-purpose type that can hold any kind of data. It is often used in libraries like NumPy to handle heterogeneous data, such as arrays containing elements of different types or ragged arrays where sub-arrays have different lengths.)
* S - string
* U - unicode string
* In Python 3.x and modern programming environments, strings are Unicode by default. This means that when you define a string in Python, it can contain characters from any language or symbol set supported by Unicode.
* Strings in these contexts can only represent characters from a limited set, often limited to the ASCII character set (128 characters).
* Unicode strings, on the other hand, are designed to handle text from all writing systems around the world.

## **Creating Arrays With a Defined Data Type**

## We use the array() function to create arrays, this function can take an optional argument: dtype that allows us to define the expected data type of the array elements

* Eg: # int

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

print(arr.dtype)

* # string

arr = np.array([1, 2, 3, 4], dtype='S')

print(arr.dtype)

## **Converting Data Type on Existing Arrays**

* The best way to change the data type of an existing array, is to make a copy of the array with the astype() method.
* The astype() function creates a copy of the array, and allows you to specify the data type as a parameter.
* The data type can be specified using a string, like 'f' for float, 'i' for integer etc. or you can use the data type directly like float for float and int for integer.
* Eg: arr=np.array([1.1, 2.1, 3.1])  
   newarr=arr.astype('i')  
   print(newarr)  
   print(newarr.dtype)

## **The Difference Between Copy and View**

* The main difference between a copy and a view of an array is that the copy is a new array, and the view is just a view of the original array.
* 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.

**Copy**

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

arr[0] = 42  
  
print(arr)  
print(x)

* The copy SHOULD NOT be affected by the changes made to the original array.

**View**

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

* The view SHOULD be affected by the changes made to the original array.

## **Check if Array Owns its Data**

* As mentioned above, copies own the data, and views does not own the data, but how can we check this?
* Every NumPy array has the attribute base that returns None if the array owns the data.
* Otherwise, the base  attribute refers to the original object.

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

* The copy returns None.
* The view returns the original array.

## **Returns Copy or View?**

Eg:

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

* returns the original array, so it is a view.

## **Shape of an Array**

* The shape of an array is the number of elements in each dimension.
* NumPy arrays have an attribute called shape that returns a tuple with each index having the number of corresponding elements.

Eg:

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

* The example above returns (2, 4), which means that the array has 2 dimensions, where the first dimension has 2 elements and the second has 4.
* Create an array with 5 dimensions using ndmin using a vector with values 1,2,3,4 and verify that last dimension has value 4.
* Eg:

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

## **Reshaping arrays**

* Reshaping means changing the shape of an array.
* The shape of an array is the number of elements in each dimension.
* By reshaping we can add or remove dimensions or change number of elements in each dimension.
* Convert the following 1-D array with 12 elements into a 2-D array.
* The outermost dimension will have 4 arrays, each with 3 elements

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

#O/p:[[1,2,3],[4,5,6],[7,8,9],[10,11,12]]

* 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

Eg:

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

#O/p: [[[1,2],[3,4],[5,6]],[[7,8],[9,10],[11,12]]]

## **Can We Reshape Into any Shape?**

* Yes, as long as the elements required for reshaping are equal in both shapes.
* We can reshape an 8 elements 1D array into 4 elements in 2 rows 2D array but we cannot reshape it into a 3 elements 3 rows 2D array as that would require 3x3 = 9 elements.

## **Flattening the arrays**

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

Eg:

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

**Other functions that commonly used for array creation**

1. Arange():

* **numpy.arange**: Create an array with a range of values.
* arr = np.arange(0, 10, 2)

print(arr)

1. Ones() :

* The numpy.ones function is used to create an array filled with ones. You can specify the shape and data type of the array.
* # Create a 2-D array of ones with shape (3, 4)

arr = np.ones((3, 4), dtype=int)

print(arr)

* # Create a 3-D array of ones with shape (2, 2, 3)

arr = np.ones((2, 2, 3))

print(arr)

1. Zeros():

* The numpy.zeros function is used to create an array filled with zeros. Similar to numpy.ones, you can specify the shape and data type of the array.
* # Create a 2-D array of zeros with shape (2, 3) and integer data type

arr = np.zeros((2, 3), dtype=int)

print(arr)

* # Create a 3-D array of zeros with shape (2, 2, 3)

arr = np.zeros((2, 2, 3))

print(arr)

1. Linspace():

* **numpy.linspace**: Create an array with linearly spaced values.
* arr = np.linspace(0, 1, 5)

print(arr)

1. diag():

* The numpy.diag function is used to extract or construct a diagonal array.
* f=np.diag([3,4,5,6,7,8])

print(f)

1. **eye():**

* **numpy.eye**: Create a 2-D array with ones on the diagonal and zeros elsewhere.
* arr = np.eye(3)

print(arr)

### Mathematical Operations

* **numpy.add**: Add two arrays element-wise.

arr = np.add(np.array([1, 2]), np.array([3, 4]))

* **numpy.subtract**: Subtract one array from another element-wise.

arr = np.subtract(np.array([1, 2]), np.array([3, 4]))

* **numpy.multiply**: Multiply two arrays element-wise.

arr = np.multiply(np.array([1, 2]), np.array([3, 4]))

* **numpy.dot**: Dot product of two arrays.

arr = np.dot(np.array([1, 2]), np.array([3, 4]))

* **numpy.sum**: Sum of array elements.

arr = np.sum(np.array([1, 2, 3, 4]))

* **numpy.mean**: Mean of array elements.

arr = np.mean(np.array([1, 2, 3, 4]))

* **numpy.std**: Standard deviation of array elements.

arr = np.std(np.array([1, 2, 3, 4]))

* **Variance (np.var)**: Calculates the variance of the elements in the array.

arr = np.var(np.array([1, 2, 3, 4]))

* **numpy.max**: Maximum value in an array.

arr = np.max(np.array([1, 2, 3, 4]))

* **numpy.min**: Minimum value in an array.

arr = np.min(np.array([1, 2, 3, 4]))

## **Iterating Arrays**

* Iterating means going through elements one by one.
* As we deal with multi-dimensional arrays in numpy, we can do this using basic for loop of python.
* If we iterate on a 1-D array it will go through each element one by one.

Eg: arr = np.array([1, 2, 3])  
 for x in arr:  
   print(x)

* In a 2-D array it will go through all the rows.

Eg:

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

* Iterate on each scalar element of the 2-D array

Eg:

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

## **Iterating Arrays Using nditer()**

* The function nditer() is a helping function that can be used from very basic to very advanced iterations. It solves some basic issues which we face in iteration

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

## **Enumerated Iteration Using ndenumerate()**

* Enumeration means mentioning sequence number of somethings one by one.
* Sometimes we require corresponding index of the element while iterating, the ndenumerate() method can be used for those use cases.

Eg:

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

* Enumerate on following 2D array's elements

Eg:

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

## **Joining NumPy Arrays**

* Joining means putting contents of two or more arrays in a single array. You're joining them along an existing dimension (length of the table)
* We pass a sequence of arrays that we want to join to the concatenate() function, along with the axis. If axis is not explicitly passed, it is taken as 0.

Eg: arr1 = np.array([1, 2, 3])  
 arr2 = np.array([4, 5, 6])  
 arr = np.concatenate((arr1, arr2))  
 print(arr)

#O/P : [1,2,3,4,5,6]

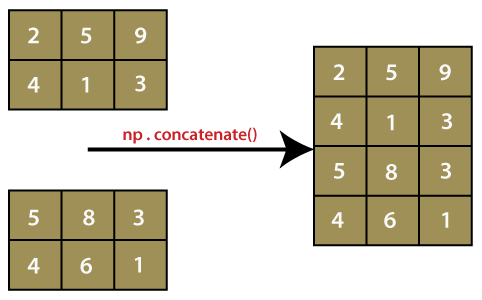
**Join two 2-D arrays along rows (axis=0)**

arr1 = np.array([[2, 5,9], [4,1,3]])

arr2 = np.array([[5,8,3], [4,6, 1]])

arr = np.concatenate((arr1, arr2))

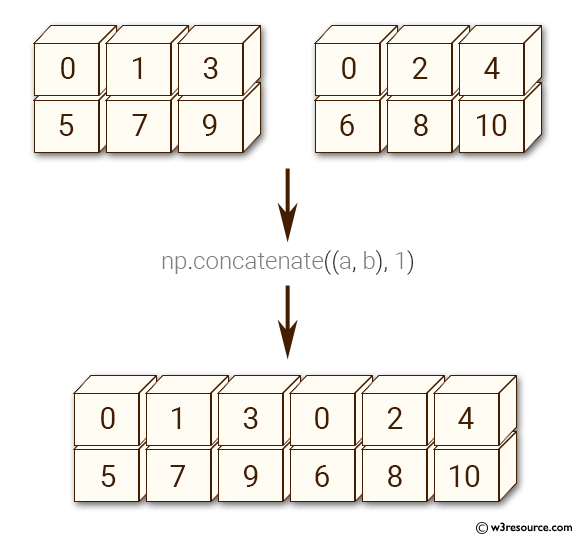
print(arr)



**Join two 2-D arrays along columns (axis=1)**:

Eg:

arr1 = np.array([[0,1, 3], [5,7, 9]])  
 arr2 = np.array([[0,2, 4], [6,8, 10]])  
 arr = np.concatenate((arr1, arr2),axis=1)  
 print(arr)



## **Joining Arrays Using Stack Functions**

* Stacking is same as concatenation, the only difference is that stacking is done along a new axis.
* Unlike the [concatenate()](https://www.pythontutorial.net/python-numpy/numpy-concatenate/) function, the stack() function joins 1D arrays to be one 2D array and joins 2D arrays to be one 3D array.
* We can concatenate two 1-D arrays along the second axis which would result in putting them one over the other, ie. stacking.
* We pass a sequence of arrays that we want to join to the stack() method along with the axis. If axis is not explicitly passed it is taken as 0.

**CHOOSING BETWEEN STACK AND CONCATENATION:**The choice depends on your desired outcome:• If you want to combine arrays with a new level of organization, use stack.• If you simply want to join arrays along an existing dimension, use concatenate.

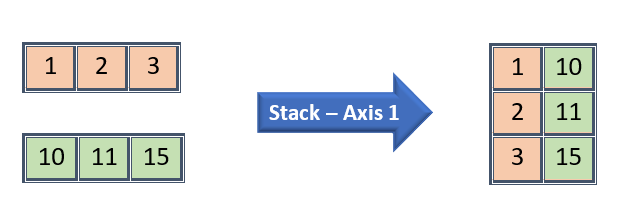
* Eg:

arr1 = np.array([1, 2, 3])  
arr2 = np.array([10, 11, 15])  
arr = np.stack((arr1, arr2), axis=1)  
print(arr)

#O/P : [[1,10],

[2,11],

[3,15]]





## **Stacking Along Rows**

* NumPy provides a helper function: hstack() to stack along rows.
* Stacks arrays along the horizontal axis (column-wise)
* **1-D Arrays:** The output is a 1-D array formed by concatenating the input arrays.
* **2-D Arrays:** The output is a 2-D array where the input arrays are concatenated along the columns.
* **Higher-Dimensional Arrays:** The output has the same number of dimensions as the input arrays, but the second axis is increased to accommodate all input arrays.
* If Axis is 0, then it will join by first dimension
* If Axis is 1, the it will join by second dimension

Eg:

**# Create two 2-D arrays**

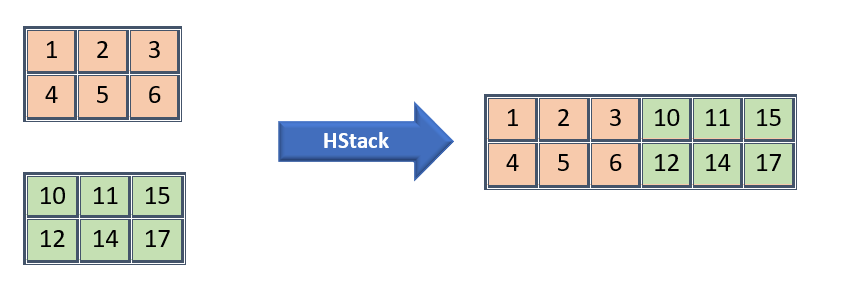
array1 = np.array([[1, 2, 3], [4, 5, 6]])

array2 = np.array([ [10, 11, 15], [12, 14,17]])

# Stack them horizontally

result = np.hstack((array1, array2))

#O/p : Stacked array: [[ 1 2 3 10 11 15] [ 4 5 6 12 14 17]]



**# Create two 3-D arrays**

array1 = np.array([ [[1, 2], [3, 4]] , [[5, 6], [7, 8]] ])

array2 = np.array([ [[ 9, 10], [11, 12]] , [[13, 14], [15, 16]] ])

# Stack them horizontally

result = np.hstack((array1, array2))

Stacked array:

[ [[ 1 2]

[ 3 4]

[ 9 10]

[11 12]]

[[ 5 6]

[ 7 8]

[13 14]

[15 16]] ]

## **Stacking Along Columns**

* NumPy provides a helper function: vstack()  to stack along columns.
* Stacks arrays along the vertical axis (row-wise).
* vstack in NumPy is a function used to stack arrays vertically, which means stacking arrays row-wise.
* This function is useful when you need to combine multiple arrays into a single array by stacking them one on top of the other.
* The arrays must have the same shape along all but the first axis.
* **When stacking 1-D arrays:** The output is a 2-dimensional array.
* **When stacking 2-D arrays:** The output remains a 2-dimensional array.
* **When stacking higher-dimensional arrays:** The output has one more dimension than the input arrays, but the first axis is increased.
* Eg:

**# Create two 1 D arrays**

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

#O/p:[[1,2,3]

[4,5,6]]

**# Create two 2-D arrays**

array1 = np.array([[1, 2, 3], [4, 5, 6]])

array2 = np.array([[7, 8, 9], [10, 11, 12]])

# Stack them vertically

result = np.vstack((array1, array2))

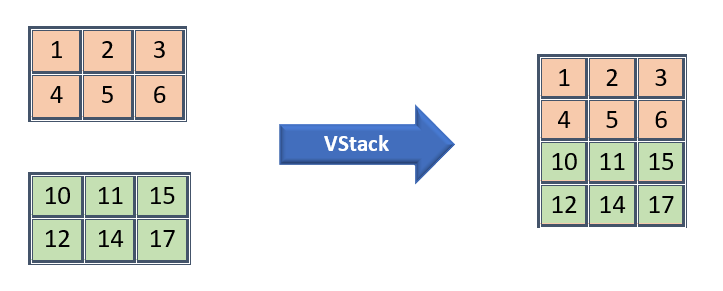
# O/p: Stacked array:

[[ 1 2 3]

[ 4 5 6]

[ 7 8 9]

[10 11 12]]



**# Create two 3-D arrays**

array1 = np.array([ [[1, 2, 3]], [[4, 5, 6]] ])

array2 = np.array([[[7, 8, 9]], [[10, 11, 12]]])

# Stack them vertically

result = np.vstack((array1, array2))

Stacked array:

[[[ 1 2 3]]

[[ 4 5 6]]

[[ 7 8 9]]

[[10 11 12]]]

Shape of stacked array: (4, 1, 3)

## **Stacking Along Height (depth)**

* NumPy provides a helper function: dstack() to stack along height, which is the same as depth
* The, np.dstack() function is used to stack these arrays horizontally along the third axis to create a 3-D array.
* This is useful when you want to combine multiple 2-D arrays into a single 3-D array by stacking them along a new depth axis.
* The output of dstack is always a 3-dimensional array, regardless of the dimensionality of the input arrays.

**# Create two 1-D arrays**

array1 = np.array([1, 2, 3])

array2 = np.array([4,5, 6])

# Stack them depth-wise

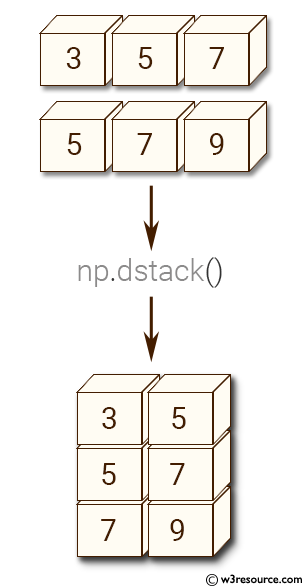
result = np.dstack((array1, array2))

#O/p:Stacked array:

[ [[1 4]

[2 5]

[3 6]] ]



**# Create two 2-D arrays**

array1 = np.array([[1, 2, 3], [4, 5, 6]])

array2 = np.array([[7, 8, 9], [10, 11, 12]])

# Stack them depth-wise

result = np.dstack((array1, array2))

#O/p:Stacked array:

[[[ 1 7]

[ 2 8]

[ 3 9]]

[[ 4 10]

[ 5 11]

[ 6 12]]]

**# Create two 3-D arrays**

array1 = np.array([[[1, 2], [3, 4]], [[5, 6], [7, 8]]])

array2 = np.array([[[9, 10], [11, 12]], [[13, 14], [15, 16]]])

#O/p : Stacked array:

[[[ 1 2 9 10]

[ 3 4 11 12]]

[[ 5 6 13 14]

[ 7 8 15 16]]]

Shape of stacked array: (2, 2, 4)

**Comparison with Other Stacking Functions**

* **vstack**: Stacks arrays vertically (row-wise).
* **hstack**: Stacks arrays horizontally (column-wise).
* **dstack**: Stacks arrays depth-wise (along the third axis)

## **Splitting NumPy Arrays**

* Splitting is reverse operation of Joining.
* Joining merges multiple arrays into one and Splitting breaks one array into multiple.
* We use array\_split() for splitting arrays, we pass it the array we want to split and the number of splits.
* If the array has less elements than required, it will adjust from the end accordingly.

#1 D array

* arr = np.array([1, 2, 3, 4, 5, 6])  
  newarr = np.array\_split(arr, 3)  
  print(newarr)

#O/p:array([1,2]),array([3,4]), array([5,6])

* arr = np.array([1, 2, 3, 4, 5, 6])  
  newarr = np.array\_split(arr, 4)  
  print(newarr)

#O/p: array([1,2]),array([3,4]),array([5]),array([6])

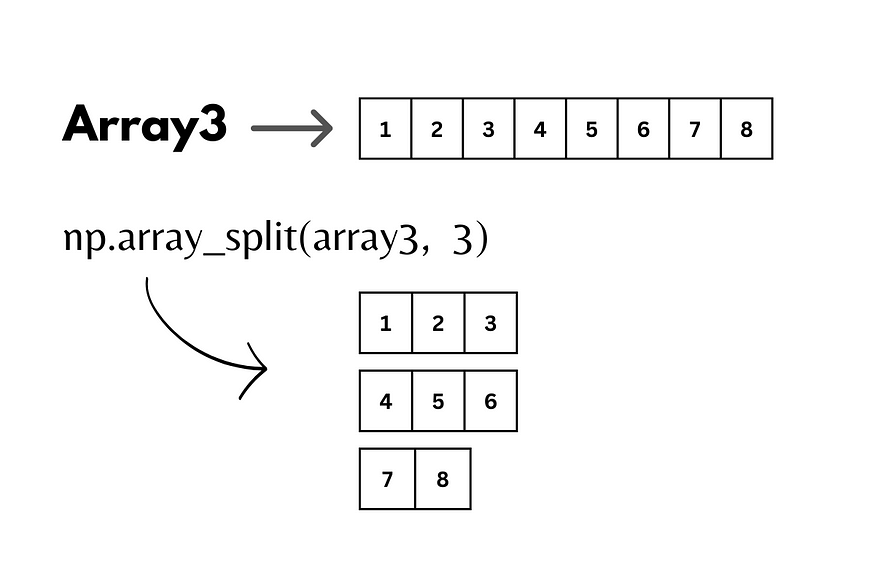
* We also have the method split() available but it will not adjust the elements when elements are less in source array for splitting like in example above, array\_split() worked properly but split() would fail.

## **Split Into Arrays**

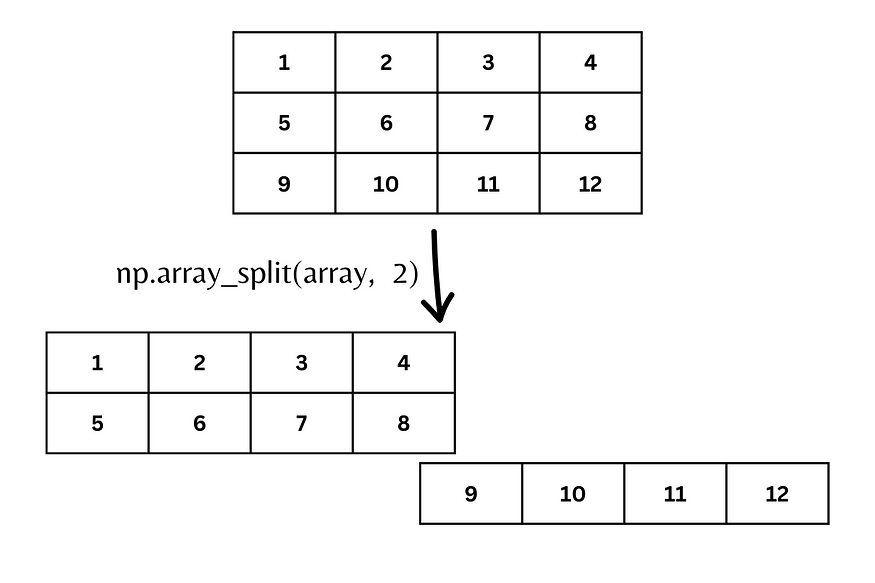
* The return value of the array\_split() method is an array containing each of the split as an array.
* If you split an array into 3 arrays, you can access them from the result just like any array element:

Eg:

arr = np.array([1, 2, 3, 4, 5, 6])  
newarr = np.array\_split(arr, 3)  
  
print(newarr[0])  
print(newarr[1])  
print(newarr[2])



**# Create a 2-D array**



* array = np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]])

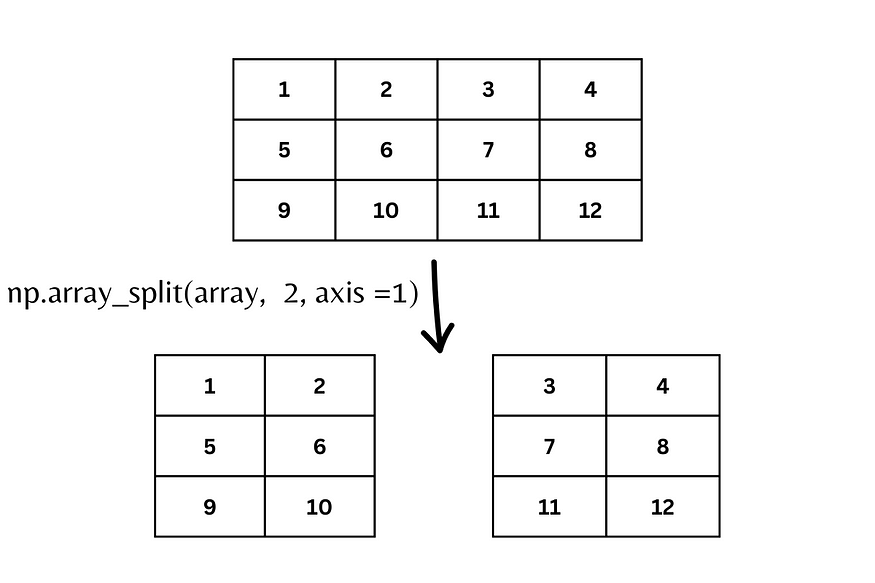
# Split the array into 2 equal sub-arrays along the second axis (columns)

result = np.split(array, 2, axis=1)

#O/p : Split array along axis 1:

[ array([[ 1, 2],[ 5, 6], [ 9, 10]]),

array([[ 3, 4], [ 7, 8], [11, 12]]) ]



* arr = np.array([[1, 2], [3, 4], [5, 6], [7, 8], [9, 10], [11, 12]])  
  newarr = np.array\_split(arr, 3)  
  print(newarr)

#O/p: [array([[1, 2],[3, 4]]),

array([[5, 6],[7, 8]]),

array([[ 9, 10], [11, 12]])]

* Split the 2-D array into three 2-D arrays along rows.
* 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)

#O/p: [array([[ 1],

[ 4],

[ 7],

[10],

[13],

[16]]), array([[ 2],

[ 5],

[ 8],

[11],

[14],

[17]]), array([[ 3],

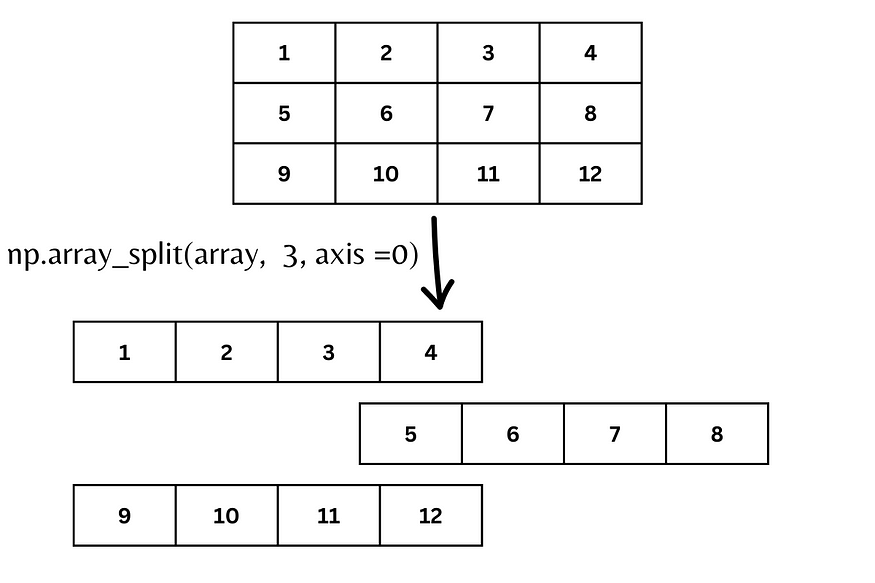
[ 6],

[ 9],

[12],

[15],

[18]])]



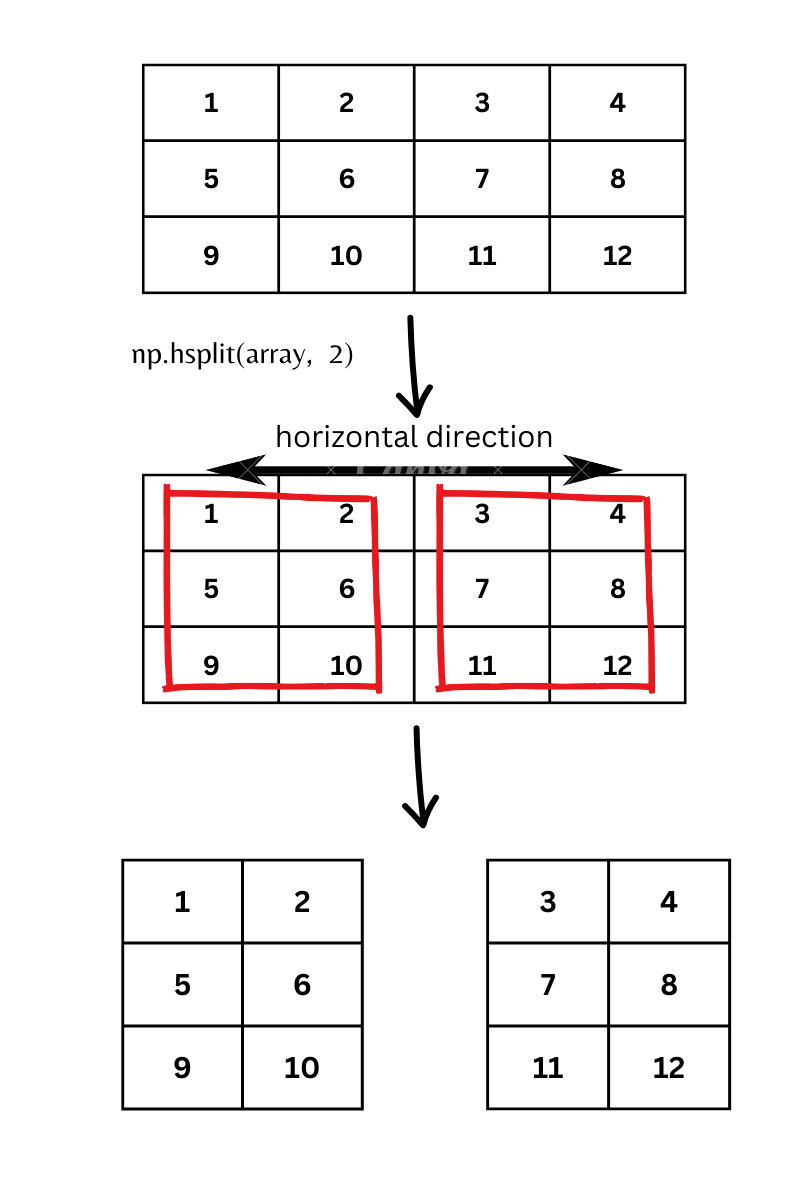
* An alternate solution is using hsplit() opposite of hstack()

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

#O/p: [array([[ 1],[ 4],[ 7],[10], [13],[16]]),

array([[ 2], [ 5],[ 8],[11],[14],[17]]),

array([[ 3],[ 6],[ 9], [12],[15],[18]])]



## **Searching Arrays**

* You can search an array for a certain value, and return the indexes that get a match.
* To search an array, use the where() method.
* For replacing a value in a array using where()
* np.where(condition, x, y)
* condition: An array-like structure that contains boolean values (True or False).
* x: The values to place in the output array where the condition is True.
* y: The values to place in the output array where the condition is False.
* Find the index where the value equal to 4
  + - arr = np.array([1, 2, 3, 4, 5, 4, 4])  
      x = np.where(arr == 4)  
      print(x)
* Find the indexes where the values are even:
  + - import numpy as np  
      arr = np.array([1, 2, 3, 4, 5, 6, 7, 8])  
      x = np.where(arr%2 == 0)  
      print(x)
* Given the array a = np.array([1, 2, 3, 4, 5]) and b = np.array([6, 7, 8, 9, 10]), use np.where to replace elements in a that are less than 3 with 0,.

# Define the arrays

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

b = np.array([6, 7, 8, 9, 10])

# Replace elements in 'a' that are less than 3 with -1

a\_modified = np.where(a < 3, -1, a)

# Horizontally stack 'a\_modified' and 'b'

result = np.hstack((a\_modified, b))

## **Search Sorted (Optional)**

* There is a method called searchsorted() which performs a binary search in the array, and returns the index where the specified value would be inserted to maintain the search order.
* The searchsorted() method is assumed to be used on sorted arrays.
* The method starts the search from the left and returns the first index where the number 7 is no longer larger than the next value.
  + - arr = np.array([6, 7, 8, 9])  
      x = np.searchsorted(arr, 7)  
      print(x)

### Search From the Right Side

* By default the left most index is returned, but we can give side='right' to return the right most index instead.
  + - arr = np.array([6, 7, 8, 9])  
      x = np.searchsorted(arr, 7, side='right')  
      print(x)
* The method starts the search from the right and returns the first index where the number 7 is no longer less than the next value.

### Multiple Values

* To search for more than one value, use an array with the specified values.
* Find the indexes where the values 2, 4, and 6 should be inserted:
  + - import numpy as np  
      arr = np.array([1, 3, 5, 7])  
      x = np.searchsorted(arr, [2, 4, 6])  
      print(x)
* #O/p: [1 2 3]

**Output Explanation:**

* 2 should be inserted at index 1 (between 1 and 3)
* 4 should be inserted at index 2 (between 3 and 5)
* 6 should be inserted at index 3 (between 5 and 7)

## **Sorting Arrays**

* Sorting means putting elements in an ordered sequence.
* Ordered sequence is any sequence that has an order corresponding to elements, like numeric or alphabetical, ascending or descending.
* The NumPy ndarray object has a function called sort(), that will sort a specified array.
* arr = np.array([3, 2, 0, 1])  
  print(np.sort(arr))

This method returns a copy of the array, leaving the original array unchanged.

* You can also sort arrays of strings, or any other data type:
* arr = np.array(['banana', 'cherry', 'apple'])  
  print(np.sort(arr))

## **# Sorting a 2-D Array**

* If you use the sort() method on a 2-D array, both arrays will be sorted:
* arr = np.array([[3, 2, 4], [5, 0, 1]])  
  print(np.sort(arr))

## **Filtering Arrays**

* Getting some elements out of an existing array and creating a new array out of them is called filtering.
* In NumPy, you filter an array using a boolean index list.
* A boolean index list is a list of booleans corresponding to indexes in the array.
* If the value at an index is True that element is contained in the filtered array, if the value at that index is False that element is excluded from the filtered array.
* arr = np.array([41, 42, 43, 44])  
  x = [True, False, True, False]  
  newarr = arr[x]  
  print(newarr)

## **Creating the Filter Array**

Create a filter array that will return only values higher than 42:

* 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)

Create a filter array that will return only even elements from the original array:

* 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)

## **Creating Filter Directly From Array**

* We can directly substitute the array instead of the iterable variable in our condition and it will work just as we expect it to.
* Create a filter array that will return only values higher than 42:
* import numpy as np  
    
  arr = np.array([41, 42, 43, 44])  
    
  filter\_arr = arr > 42  
    
  newarr = arr[filter\_arr]  
    
  print(filter\_arr)  
  print(newarr)
* Create a filter array that will return only even elements from the original array:
* import numpy as np  
    
  arr = np.array([1, 2, 3, 4, 5, 6, 7])  
    
  filter\_arr = arr % 2 == 0  
    
  newarr = arr[filter\_arr]  
    
  print(filter\_arr)  
  print(newarr)

### Random Numbers

* Random numbers are numbers that are generated in such a way that each number has an equal probability of being chosen within a specified range.
* **Random Numbers** are used in a wide range of applications, from scientific simulations to cryptography (including key generation, encryption, digital signatures, and secure communication protocols) and gaming (
* Random Events (The locations and times at which enemies appear can be controlled by random numbers to keep gameplay unpredictable and engaging)
* Randomized Outcomes (Shuffling a deck of cards in digital card games relies on random numbers to ensure fair play and unpredictability).)
* However, not all random numbers are created equal. They can be broadly categorized into two types:
* pseudo-random numbers
* True random numbers.

### True Random Numbers

**True Random Numbers** are generated from physical processes that are inherently unpredictable, like radioactive decay or atmospheric noise.

* **Unpredictable:** Their next value can't be guessed.
* **Examples:** Flipping a coin, rolling a dice, measuring radioactive decay.

**Pseudo-Random Numbers**

**Pseudo-Random Numbers** are generated using algorithms and a starting value called a seed. They appear random, but are actually predictable if you know the seed.

* **Deterministic:** If you start with the same seed, you get the same sequence of numbers.
* **Examples:** Random numbers generated by computer algorithms like the Linear Congruential Generator (LCG).
* **True Random Numbers:** Best for cryptographic applications because they are truly unpredictable.
* **Pseudo-Random Numbers:** Suitable for non-cryptographic applications where high speed and reproducibility are important.

**Random Number Generation in NumPy**

* Python defines a set of functions that are used to generate or manipulate random numbers through the random module.
* Functions in the random module rely on a pseudo-random number generator function random(), which generates a random float number between 0.0 and 1.0.
* This particular type of functions is used in a lot of games, lotteries, or any application requiring a random number generation.

#### **Basic Random Number Generation**

1. **Single Random Float**

**import numpy as np**

**random\_float = np.random.rand()**

**print(random\_float) # Generates a random float between 0 and 1**

1. **Array of Random Integers:**

**random\_integers = np.random.randint(10, 50, size=5)**

**print(random\_integers)**

**# Generates an array of 5 random integers between 10 and 50**

#### **Random Arrays**

1. **2D Array of Random Floats:**

random\_2d\_array = np.random.rand(3, 3)

print(random\_2d\_array)

# Generates a 3x3 array of random floats between 0 and 1

**Using random.uniform**

# Generate a single random float between 5 and 10 random\_float\_uniform = np.random.uniform(5, 10) print(random\_float\_uniform)

# Generate an array of 10 random floats between 5 and 10 random\_floats\_uniform = np.random.uniform(5, 10, 10) print(random\_floats\_uniform)

1. **1D Array from Normal Distribution:**

random\_normal = np.random.randn(10)

print(random\_normal)

# Generates a 1D array of 10 random numbers from a standard normal distribution

#### **Random Choice**

1. **Without Replacement:**

random\_sample =np.random.choice([10, 20, 30, 40, 50], size=5, replace=False)

print(random\_sample)

# Generates a random sample of 5 elements without replacement

1. **With Replacement:**

random\_sample\_with\_replacement = np.random.choice([10, 20, 30, 40, 50], size=3, replace=True)

print(random\_sample\_with\_replacement)

# Generates a random sample of 3 elements with replacement

#### **Setting a Seed**

* Setting a seed ensures reproducibility of random number sequences:

np.random.seed(42)

reproducible\_random\_array = np.random.rand(5)

print(reproducible\_random\_array)

# The same array will be generated every time this code is run

#### **Shuffling**

1. **Shuffle Array Elements:**

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

np.random.shuffle(array)

print(array)

# Shuffles the elements of the array in-place

1. **Shuffle Rows of a 2D Array:**

array\_2d = np.array([[1, 2], [3, 4], [5, 6]])

np.random.shuffle(array\_2d)

print(array\_2d)

# Shuffles the rows of the 2D array

#### **Statistical Distributions**

1. **Uniform Distribution:**

**Uniform Distribution** is a type of probability distribution where all outcomes are equally likely. Each value within a specified range has an equal probability of occurring. This distribution is often used when there is no reason to prefer one outcome over another.

uniform\_random\_numbers = np.random.uniform(-1, 1, size=1000)

# Generate 1000 random floats between -1 and 1

1. **Poisson Distribution:**

**Poisson Distribution** is a probability distribution that models the number of events occurring within a fixed interval of time or space, given the events happen with a known constant rate and are independent of the time since the last event. It is often used to model rare events.

poisson\_random\_numbers = np.random.poisson(lam=5, size=1000)

# Generate 1000 random numbers from a Poisson distribution with lambda = 3

**Installing jupyter notebook in. ipynb format**

**! pip install nbconvert**

**! jupyter nbconvert --to html /content/your\_notebook.ipynb**