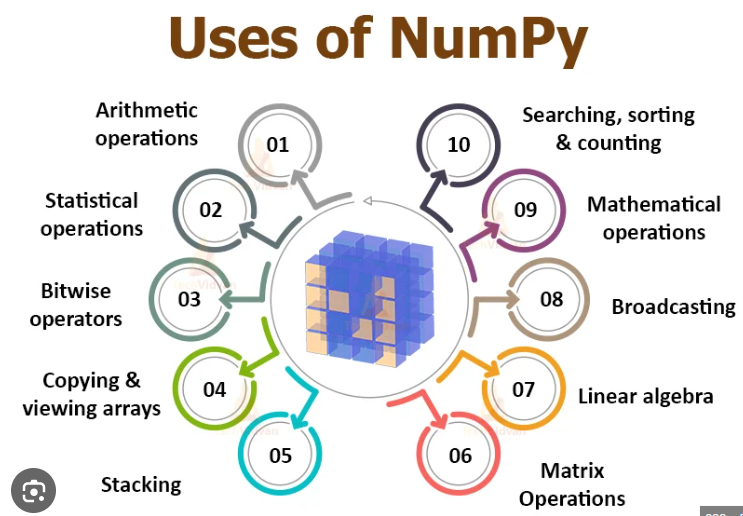
Introduction to NumPy:

NumPy is a Python library used for working with arrays. It also has functions for working in domain of linear algebra, Fourier transform, and matrices. NumPy was created in 2005 by Travis Oliphant. It is an open-source project and you can use it freely. NumPy stands for Numerical Python. 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.

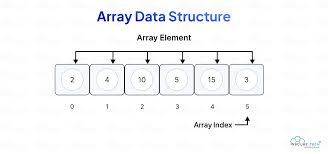
Array:

An array is defined as a collection of items of similar data type that are stored at contiguous memory locations. Arrays in Python are Data Structures that can hold multiple values of the same type.

Arrays are also mutable ( change ) and not fixed in size, which means they can grow and shrink throughout the life of the program. Items can be added and removed, making them very flexible to work with.

Accessing array elements in Python :

Each and every element in an array is given an index or subscript. To access array elements, you need to specify the index values. Array Indexing starts at 0 .



NumPy Array

NumPy is a special Python library with a large collection of high-level mathematical functions to operate on arrays. It also has functions for working with statistical calculations, linear algebra, fourier transform, and matrices.

Example:

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

Example:

import numpy as np

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

print(arr)

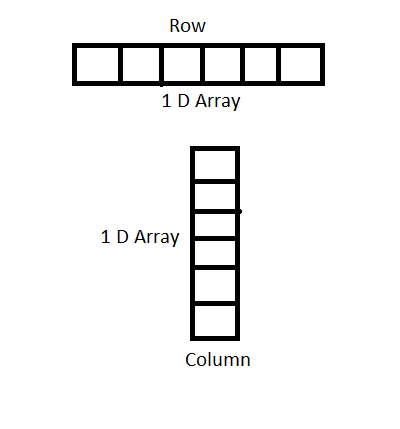
0-D arrays, or Scalars, are the elements in an array. Each value in an array is a 0-D array.

Example:

import numpy as np  
arr = np.array(42)  
print(arr)

1-D Arrays

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.



Example:

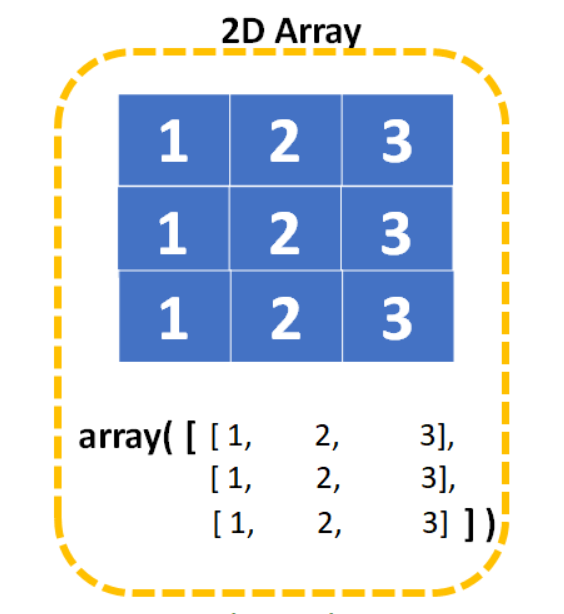
import numpy as np

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

print(arr)

2-D Arrays

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.



Example:

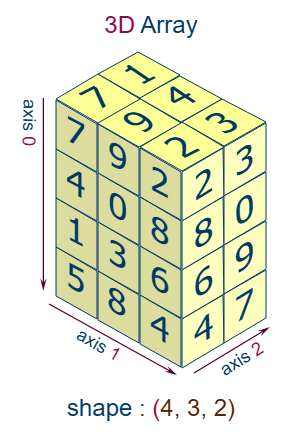
import numpy as np

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

print(arr)

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.



Example:

import numpy as np

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

print(arr)

Higher Dimensional Arrays

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

Example:

import numpy as np

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

print(arr)

print('number of dimensions :', arr.ndim)

Indexing and Slicing:

Indexing refers to accessing individual elements or groups of elements within an array. NumPy supports various types of indexing, including integer indexing, boolean indexing, and advanced or fancy indexing. For example, using integer indexing, you can access specific elements by their position

Example:

import numpy as np

arr = np.array([10, 20, 30, 40, 50])

first\_element = arr[0]# Output: 10

last\_element = arr[-1]# Output: 50

Slicing involves accessing a range or subset of elements within an array. This is particularly useful when working with large datasets where you need to focus on specific portions of the data. The slicing syntax in NumPy follows the pattern start:stop:step, allowing you to define the range and interval of elements to select:​

Example:

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

subset = arr[2:6]

#Output: array([2, 3, 4, 5])

subset\_step = arr[1:8:2]

#Output: array([1, 3, 5, 7])

For multi-dimensional arrays, indexing and slicing can be applied across multiple axes. For instance, in a 2D array (matrix), you can access specific rows, columns, or submatrices:

Example:

# Create a 2D array

arr\_2d = np.array([[0, 1, 2], [3, 4, 5], [6, 7, 8]])

# Access the element at row 1, column 2

element = arr\_2d[1, 2] # Output: 5

# Slice the first two rows and all columns

subset\_rows = arr\_2d[:2, :] # Output: array([[0, 1, 2],

# [3, 4, 5]])

# Slice all rows and the first two columns

subset\_cols = arr\_2d[:, :2] # Output: array([[0, 1],

# [3, 4],

# [6, 7]])

For a comprehensive understanding of NumPy's indexing and slicing capabilities, the following resources are highly recommended

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

Example:

import numpy as np

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

print(arr[0])

Output:

1

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.

Example:

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])

Ouput: 2nd element on 1st dim: 2

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.

Example

import numpy as np

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

print(arr[0, 1, 2])

Output: 6

Negative Indexing

Use negative indexing to access an array from the end.

Example

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])

Output: Last element from 2nd dim: 10

Concatenating Arrays:

Array concatenation is a fundamental operation in NumPy that allows you to join two or more arrays along a specified axis. The np.concatenate() function is commonly used for this purpose. It works on arrays of the same shape, except in the dimension corresponding to the axis along which they are joined. For example, one-dimensional arrays can be combined end-to-end, while two-dimensional arrays can be joined by rows or columns. In addition to concatenate(), NumPy provides convenience functions like vstack() for vertical stacking and hstack() for horizontal stacking. When a new dimension is needed, np.stack() is used, which creates a new axis in the resulting array. These operations are widely used in data processing, machine learning, and scientific computing where merging datasets or results is often required.

Example:

import numpy as np

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

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

result = np.concatenate((a, b))

print(result)

Output: # [1 2 3 4 5 6]

**2. Concatenate 2D Arrays**

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

b = np.array([[5, 6]])

# Axis 0 → Add rows

print(np.concatenate((a, b), axis=0))

# [[1 2]

# [3 4]

# [5 6]]

# Axis 1 → Add columns

c = np.array([[7], [8]])

print(np.concatenate((a, c), axis=1))

Output: # [[1 2 7]

# [3 4 8]]

**3. Using np.vstack() and np.hstack():**

Vertical Stack (rows):

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

b = np.array([3, 4])

print(np.vstack((a, b)))

Output : # [[1 2]

# [3 4]]

Horizontal Stack (columns):

print(np.hstack((a, b)))

Output: # [1 2 3 4]

4. Using np.stack()

Unlike concatenate, stack() adds a new dimension.

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

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

print(np.stack((a, b)))

Output :# [[1 2 3]

# [4 5 6]]

print(np.stack((a, b), axis=1))

Output:# [[1 4]

# [2 5]

# [3 6]]

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.

Reshape From 1-D to 2-D

Example:

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)

Ouput:

[[ 1 2 3]

[ 4 5 6]

[ 7 8 9]

[10 11 12]]

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.

Example:

Split the array in 3 parts:

import numpy as np

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

newarr = np.array\_split(arr, 3)

print(newarr)

Statistical perations on Arrays:

Statistical operations help summarize and understand the distribution, spread, and tendencies of numerical data. NumPy makes this super easy!

**1. Mean (Average)**

import numpy as np

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

print(np.mean(a)) # 3.0

**2. Median**

print(np.median(a)) # 3.0

**3. Standard Deviation**

print(np.std(a)) # 1.4142...

**4. Variance**

print(np.var(a)) # 2.0

**5. Min and Max**

print(np.min(a)) # 1

print(np.max(a)) # 5

**6. Percentile**

print(np.percentile(a, 50)) # 3.0

**7. Sum and Product**

print(np.sum(a)) # 15

print(np.prod(a)) # 120

**8. Statistical Ops on Multi-D Arrays**

Example:

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

print(np.mean(b, axis=0)) # Column-wise mean → [2.5 3.5 4.5]

print(np.mean(b, axis=1)) # Row-wise mean → [2. 5.]

axis=0: across columns

axis=1: across rows

**Summary Table**

Function Purpose

np.mean() Average value

np.median() Middle value

np.std() Standard deviation

np.var() Variance

np.min() Minimum value

np.max() Maximum value

np.percentile() Value at a given percentile

np.sum() Sum of elements

np.prod() Product of elements

Data handling Using pandas:

Pandas is one of the most powerful and widely used libraries in Python for data manipulation and analysis. It provides easy-to-use data structures and functions that make working with structured data (like tables or spreadsheets) very efficient and intuitive.The two main data structures in Pandas are:

Series:

A Pandas Series is a one-dimensional labeled array that can hold data of any type such as integers, floats, strings, or even Python objects. It is similar to a list or a column in a spreadsheet, but with additional functionality. Each element in a Series is associated with an index, which allows for efficient and intuitive data access. Series are commonly used when you want to represent a single column of data or when you’re dealing with a simple list-like dataset with labels. The default index is integer-based (0, 1, 2…), but you can also specify custom index labels.

**Example 1: Creating a Simple Series**

import numpy as np

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

print(np.mean(a)) # 3.0

print(np.median(a)) # 3.0

print(np.std(a)) # 1.4142...

print(np.var(a)) # 2.0

print(np.min(a)) # 1

print(np.max(a)) # 5

print(np.percentile(a, 50)) # 3.0

print(np.sum(a)) # 15

print(np.prod(a)) # 120

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

print(np.mean(b, axis=0)) # Column-wise mean → [2.5 3.5 4.5]

print(np.mean(b, axis=1)) # Row-wise mean → [2. 5.]

axis=0: across columns

axis=1: across rows

data = [10, 20, 30, 40]

import pandas as pd

# Creating a Series from a list

data = [10, 20, 30, 40]

s = pd.Series(data)

print(s)

Output:

0 10

1 20

2 30

3 40

Default index is used (0, 1, 2, 3).

**Example 2: Series with Custom Index**

data = [100, 200, 300]

index = ['Math', 'Science', 'English']

s = pd.Series(data, index=index)

print(s)

Output:

Math 100

Science 200

English 300

Custom labels used instead of default numeric index.

**Example 3: Accessing Series Elements**

print(s['Math']) # 100

print(s[1])

OUTPUT: # 200

**Example 4: Basic Operations on Series**

s = pd.Series([1, 2, 3, 4])

print(s + 10) # Adds 10 to each element

print(s \* 2) # Multiplies each element by 2

Pandas Series are simple yet powerful. They behave like both arrays and dictionaries, allowing for quick mathematical operations and label-based indexing. They form the foundation of DataFrames (which are collections of Series).

DataFrame:

A Pandas DataFrame is a two-dimensional, tabular data structure that consists of rows and columns, very similar to a spreadsheet or SQL table. Each column in a DataFrame is a Pandas Series, and the columns can hold different types of data (integers, floats, strings, etc.). DataFrames are extremely powerful for organizing, analyzing, and manipulating structured data. You can load data from various file formats (CSV, Excel, SQL, etc.), filter rows, select specific columns, apply calculations, and more — all with very readable code.

**Example 1: Creating a DataFrame from a Dictionary**

import pandas as pd

data = {

'Name': ['Alice', 'Bob', 'Charlie'],

'Age': [25, 30, 35],

'City': ['New York', 'London', 'Paris']

}

df = pd.DataFrame(data)

print(df)

Output:

Name Age City

0 Alice 25 New York

1 Bob 30 London

2 Charlie 35 Paris

**Example 2: Accessing Columns and Rows**

# Access a single column

print(df['Name'])

# Access a row by index

print(df.loc[0]) # By label/index name

print(df.iloc[1]) # By numerical position

**Example 3: Adding a New Column**

df['Score'] = [85, 90, 88]

print(df)

**Example 4: Filtering Data**

# Filter rows where Age > 28

filtered = df[df['Age'] > 28]

print(filtered)

Pandas DataFrames are the go-to structure for handling structured data in Python. Whether you're working with datasets from files or manipulating in-memory data, DataFrames provide powerful tools for filtering, transforming, and analyzing data easily and efficiently.

Importing and Exporting Data between CSV Files and DataFrames

One of the most common tasks in data analysis is working with CSV (Comma-Separated Values) files. Pandas makes it extremely easy to import data from a CSV file into a DataFrame and export a DataFrame back to a CSV file. This allows you to analyze, manipulate, and visualize data using Python, and then save your results or cleaned data for later use. The functions pd.read\_csv() and DataFrame.to\_csv() are commonly used for these tasks. Pandas handles missing values, column headers, and custom delimiters, making it a robust tool for working with CSV files.

Importing Data from a CSV File

**Example:**

import pandas as pd

# Read data from a CSV file

df = pd.read\_csv('data.csv')

# Display the first 5 rows

print(df.head())

**Example 2: Exporting Data to a CSV File**

# Save the DataFrame to a new CSV file

df.to\_csv('output.csv', index=False)

With just a single line of code, you can easily load or save data between CSV files and Pandas DataFrames. This makes it super convenient to work with real-world datasets stored in CSV format, clean them up, analyze them, and share results.

Pandas Series vs NumPy ndarray

Both Pandas Series and NumPy ndarray are used for handling one-dimensional arrays of data in Python, but they serve slightly different purposes and have different features. A NumPy ndarray is a fast, flexible container for numerical data that supports vectorized operations. A Pandas Series, on the other hand, builds on top of NumPy arrays and adds labels (indices) to each element, allowing for more intuitive data handling, especially when working with real-world datasets where labeled data is common. Series is ideal for situations where row labels (like names, dates, or categories) matter, while NumPy arrays are better for raw numerical computations.

| **Feature** | **Pandas Series** | **NumPy ndarray** |
| --- | --- | --- |
| **Data Structure** | 1D labeled array | N-dimensional array |
| **Indexing** | Label-based and position-based | Position-based only |
| **Data Type Support** | Any type (int, float, string, etc.) | Usually numerical types |
| **Built-in Labels** | ✅ Yes (index) | ❌ No |
| **Missing Data Handling** | Easy with NaN, isna(), etc. | Requires manual handling |
| **Performance** | Slightly slower than NumPy | Very fast and optimized |
| **Integration** | Built on top of NumPy | Core numerical library |
| **Use Case** | Data analysis, labeled data | Scientific computing, raw math |