What is NumPy?

- NumPy is a powerful library in Python designed for efficient operations on large datasets, particularly for numerical data in the form of arrays.
- The primary data structure in NumPy is called **ndarray** (n-dimensional array).
- NumPy arrays are fundamental for performing a wide range of numerical tasks in fields such as scientific computing, machine learning, and data analysis.
- The library allows easy manipulation and mathematical operations on arrays, making it a key tool for data scientists and researchers.

Why Use NumPy Arrays?

There are several key advantages to using NumPy arrays over traditional Python lists:

1. **Faster Computation**: NumPy arrays offer efficient memory management and vectorized operations, leading to faster computations.
2. **Memory Efficiency**: NumPy arrays are more memory-efficient, especially when handling large datasets.
3. **Convenient Mathematical Operations**: It provides a rich set of mathematical functions for data manipulation and analysis.
4. **Support for High-Dimensional Data**: NumPy can handle n-dimensional arrays, which makes it ideal for complex datasets.
5. **Integration with Other Libraries**: NumPy serves as the foundation for many other scientific computing and machine learning libraries, such as Pandas, Matplotlib, and Scikit-learn.

NumPy arrays are essential for efficient numerical computation in Python, offering better performance and more flexibility compared to traditional Python data structures like lists.

Array Key Concepts

1. **1D Array**: A single row or column of elements, accessed by a single index.
2. **2D Array**: A matrix (table) with rows and columns, accessed by two indices (row, column).
3. **nD Array**: Generalized arrays with multiple dimensions (3D, 4D, etc.), accessed by multiple indices corresponding to different axes.

How to Access NumPy Arrays

- **Indexing**: Elements are accessed by indices, starting from 0.

Operation	syntax
Access single element -1D	Array[i]
Access single element -2D	Array[row,column]
Access single element -ND	Array[depth,row,column]
Binary Indexing	Array[condition]
Slicing 1D array	Array[start:stop:step]
Slicing 2D array	Array[row_start:row_stop, column_start:column_stop]
Slicing ND array	Array[depth_start:depth:stop,row_start:row_stop,col_start:col_stop]
Reverse array	[::-1]

Creating NumPy Arrays from Different Data Sources

- **Basic Arrays**: np.array()
- **Random Arrays**: np.random.rand(), np.random.randint()
- **Predefined Arrays**: np.zeros(), np.ones(), np.eye()
- **Arrays from Files**: np.loadtxt()
- **Using Ranges**: np.arange(), np.linspace()

Array Operations

NumPy supports a range of operations for array manipulation:

1. **Basic Mathematical Operations**:
    - a + b, a - b, a * b, a / b
2. **Element-wise Operations**:
    - Squaring: a**2, Modulus: a % 3
3. **Scalar Operations**:
    - Adding a scalar: a + 10, Multiplying a scalar: a * 10
4. **Math Functions/Universal Functions**:
    - Example: np.sin(a), np.sqrt(a), np.log(a)
5. **Aggregation**:
    - Example: np.sum(a), np.mean(a), np.min(a)

Operation type	**example**	**Description**	**code**
Basic mathematical	a+b, a-b,a*b,a/b	Adds/multiplies	a=np.array([1,2,3,4,5]) b=np.array([10,20,30,40,50]) print(a+b) print(b-a) print(a*b) print(a/b)
Element-wise arithmetic	a**2,a%3	Perform operation on each elements	print(a**2) #square each element print(a%3) #modulus operation print(a//3) #floor division

Scalar operations	a+10, a*10	Applies operation to all elements	print(a+5) print(a*5)

Math functions / Universal functions	np.sin(a), np.cos(a),np.sqrt(a), np.exp(a),np.log(a)	Applies function element-wise	print(np.sqrt(a)) #square root
			print(np.exp(a)) #exponential (e^x)
			print(np.log(a)) #logaritham
			print(np.sin(a)) #sine fun
			print(np.cos(a)) #cos

Aggregations	np.sum(a), np.mean(a),np.min(a), np.max(a)	Computes Single Values	print(np.sum(a)) # sum of all elements
			print(np.mean(a))
			print(np.median(a))
			print(np.min(a))
			print(np.max(a))

Axis-based aggeregation	np.sum(a, axis=0)	Aggregation along rows/column Axis=0 – column Axis=1--row	matrix=np.array([[1,2,3],[4,5,6],[7,8,9]]) print(np.sum(matrix, axis=0)) # sum of all columns print(np.sum(matrix, axis=1)) # sum of all rows
Boolean filtering	a[a>3]	Filters elements based on condition	arr = np.array([1, 2, 3, 4, 5]) print(arr > 3) # boolean array print(arr[arr > 3]) # filter elements greater than 3

Broadcasting

NumPy handles operations on arrays of different shapes through **broadcasting**.

The broadcasting rules are:

1. If the two arrays have different dimensions, NumPy automatically expands the smaller array’s shape.
2. If a dimension has size 1, NumPy duplicates its values along that dimension to match the larger array.

Example:

```
arr1 = np.array([1, 2, 3]) # Shape (3,)
arr2 = np.array([[10], [20], [30]]) # Shape (3,1)
result = arr1 + arr2
```

How it works:

- **Shape of arr1:** (3,) — A row vector [1, 2, 3]
- **Shape of arr2:** (3,1) — A column vector

Broadcasted shapes:

- arr1 becomes [[1, 2, 3], [1, 2, 3], [1, 2, 3]] — Shape (3,3)
- arr2 becomes [[10, 10, 10], [20, 20, 20], [30, 30, 30]] — Shape (3,3)

Array Manipulation

NumPy offers several functions to manipulate arrays:

Operation	Function	Example Code	Description	Note
Reshape	reshape()	arr.reshape(2, 3) → Converts (6,) → (2,3)	Change shape without modifying data	Rules for reshape(): 1.The total number of elements must remain the same. 2. Use -1 to let NumPy automatically infer a dimension:
Flatten	ravel()	arr.ravel() → [1 2 3 4 5 6]	Convert multi-dimensional array to 1D	arr.flatten() also works but returns a copy, while ravel() returns a view (changes reflect in original).
Transpose	T or transpose()	arr.T → Converts (2,3) → (3,2)	Swap rows and columns	
Swap Axes	swapaxes()	arr.swapaxes(0, 1)	Swap any two axes	Useful for multi-dimensional arrays.
Resize	resize()	arr.resize((2,4))	Change array size, truncating if needed	Unlike -- reshape(), resize() modifies the original array.
Expand Dim	expand_dims()	np.expand_dims(arr, axis=0)	Add a new axis (useful in machine learning)	Alternative: arr[:, np.newaxis] adds a new axis
Squeeze	squeeze()	arr.squeeze()	Remove dimensions of size 1	If an array has a dimension of size 1, squeeze() removes it and Useful for removing unnecessary dimensions in deep learning.
Change Type	astype()	arr.astype(int)	Convert data type	
Join Arrays	concatenate()	np.concatenate((arr1, arr2), axis=0)	Merge arrays	Use axis=1 to join along columns and axis=0 for rows
Split Arrays	split()	np.split(arr, 3)	Split into sub-arrays	Use vsplit() and hsplit() for 2D arrays.

Loading and Saving Data with NumPy

NumPy allows you to easily load and save data to files:

Operation	Function	File Format	Example Code
Save Single Array	save()	Binary .npy	np.save("data.npy", arr)
Save Multiple Arrays	savez()	Compressed .npz	np.savez("data.npz", a=arr1, b=arr2)
Save as Text File	savetxt()	.txt / .csv	np.savetxt("data.txt", arr, delimiter=",")
Load Binary File	load()	.npy	arr = np.load("data.npy")
Load Multiple Arrays	load()	.npz	data = np.load("data.npz") print(data["a"])
Load Text/CSV File	loadtxt()	.txt / .csv	arr = np.loadtxt("data.txt", delimiter=",")

What is Pandas?

Pandas is an open-source Python library for data manipulation and analysis, built on top of NumPy. It is primarily used for working with structured data, such as tabular data, and provides two main data structures:

- **Series**: A 1-dimensional labeled array (like a single column).
- **DataFrame**: A 2-dimensional labeled table (like a spreadsheet).

Why Use Pandas?

Pandas simplifies data manipulation and analysis:

1. **Efficient Data Handling**: Optimized for large datasets.
2. **Data Cleaning**: Handling missing data, duplicates, and outliers.
3. **Data Analysis**: Powerful filtering, grouping, and aggregation.
4. **Integrated with Other Libraries**: Works well with NumPy, Matplotlib, and others.
5. **Flexible Input and Output**: Supports reading/writing data from various formats (CSV, Excel, etc.).
6. **Time Series Support**: Built-in functions for working with date-time data.

Pandas Data Structures

1. **Series (1D)**:
   - Like a column in a DataFrame, it holds any data type (int, float, string).
   - Supports indexing and slicing.
2. **DataFrame (2D)**:
   - Like a table, with rows and columns.
   - Stores heterogeneous data types.
   - Each column is a Pandas Series.

Feature	Series (1D)	DataFrame (2D)
Definition	One-dimensional labeled array	Two-dimensional table with rows & columns
Structure	Like a single column (Excel, SQL)	Like a full table (Excel, SQL)
Indexing	Single index	Row and column index
Data Type	Single type (int, float, string)	Multiple types (int, float, string, etc.)
Use Case	Storing a single list of values with labels	Storing structured tabular data
Creation	pd.Series(data)	pd.DataFrame(data)

Data Selection and Indexing in Pandas

Operation	Method	Example Code
Selecting Data in a Series		
Select a Single Value (by Label)	series[label]	series["a"]
Select Multiple Values (by Labels)	series[[label1, label2]]	series[["a", "c"]]
Select Value by Position	series.iloc[position]	series.iloc[0]
Select Multiple Values by Position	series.iloc[start:end]	series.iloc[0:2]
Filter Values (Condition)	series[series > x]	series[series > 20]
Selecting Data in a DataFrame		
Select a Column	df["col"]	df["Name"]
Select Multiple Columns	df[["col1", "col2"]]	df[["Name", "Salary"]]
Select Row by Label	df.loc[label]	df.loc["b"]
Select Multiple Rows by Label	df.loc[[label1, label2]]	df.loc[["a", "c"]]
Select Row by Position	df.iloc[position]	df.iloc[0]
Select Multiple Rows by Position	df.iloc[start:end]	df.iloc[0:2]
Filter Rows (Condition)	df[df["col"] > x]	df[df["Age"] > 28]
Filter Multiple Conditions	& (AND), `	` (OR)
Modifying Data		
Add New Column	df["new_col"] = ...	df["Bonus"] = df["Salary"] * 0.10
Modify a Value	df.loc[row, col] = value	df.loc["b", "Salary"] = 65000
Drop Column	df.drop(columns=["col"])	df.drop(columns=["Bonus"], inplace=True)
Drop Row	df.drop(index=row_index)	df.drop(index="b", inplace=True)

Data Cleaning with Pandas

- **Missing Data**: Use df.isnull() to check for missing values and df.fillna() to fill missing data

Task	Method	Example Code
Check for Missing Values	df.isnull()	df.isnull().sum()
Drop Rows with Missing Values	df.dropna()	df.dropna(inplace=True)
Drop Columns with Missing Values	df.dropna(axis=1)	df.dropna(axis=1, inplace=True)
Fill Missing Values with a Specific Value	df.fillna(value)	df.fillna(0, inplace=True)
Fill with Mean	df.fillna(df["col"].mean())	df["Age"].fillna(df["Age"].mean(), inplace=True)
Fill with Median	df.fillna(df["col"].median())	df["Salary"].fillna(df["Salary"].median(), inplace=True)
Fill with Mode	df.fillna(df["col"].mode()[0])	df["Age"].fillna(df["Age"].mode()[0], inplace=True)
Forward Fill (ffill)	df.fillna(method="ffill")	df.fillna(method="ffill", inplace=True)
Backward Fill (bfill)	df.fillna(method="bfill")	df.fillna(method="bfill", inplace=True)

- **Handling Duplicates**: Use df.drop_duplicates() to remove duplicate rows

Task	Method	Example Code
Check for Duplicates	df.duplicated()	df.duplicated().sum()
Remove Duplicate Rows	df.drop_duplicates()	df.drop_duplicates(inplace=True)
Remove Duplicates Based on a Column	df.drop_duplicates(subset=["col"])	df.drop_duplicates(subset=["Name"], inplace=True)

- **Outliers**: Handle outliers using the Interquartile Range (IQR) method.

Task	Method	Example Code
Calculate Q1, Q3, and IQR	IQR formula	Q1 = df["col"].quantile(0.25), Q3 = df["col"].quantile(0.75), IQR = Q3 - Q1
Find Outliers Using IQR	(col < lower_bound) or (col > upper_bound)	`df[(df["col"] < Q1 - 1.5*IQR)
Remove Outliers	Filtering within bounds	df = df[(df["col"] >= lower_bound) & (df["col"] <= upper_bound)]
Replace Outliers with Median	df["col"].mask(condition, median)	df["Salary"] = df["Salary"].mask(df["Salary"] > upper_bound, df["Salary"].median())

Merging and Joining DataFrames

Operation	Function	Purpose	Syntax Example
Merging	pd.merge()	Combine DataFrames based on common column(s).	pd.merge(df1, df2, on='ID', how='inner')
Joining	df.join()	Combine DataFrames based on index.	df1.join(df2, how='left')
Concatenation	pd.concat()	Stack DataFrames vertically or horizontally.	pd.concat([df1, df2], axis=0)

Time Series Data with Pandas

Pandas provides powerful tools for working with time series data, such as:

- **date_range()**: Generate a range of dates.
- **set_index()**: Set a date column as the index.
- **resample()**: Resample data at different frequencies (e.g., monthly).
- **shift()**: Shift data by a specific number of periods.

Operation	Function/Method	Example
Create a Time Series	pd.date_range()	pd.date_range('2025-01-01', periods=5, freq='D')
Convert to DateTime	pd.to_datetime()	pd.to_datetime(df['Date'])
Set Date as Index	df.set_index('Date')	df.set_index('Date', inplace=True)
Resample Data	df.resample()	df.resample('M').mean()
Shift Data	df.shift()	df.shift(1)
Rolling Calculations	df.rolling()	df['Value'].rolling(window=3).mean()
Fill Missing Data	df.fillna()	df.fillna(method='ffill')
Plot Time Series	df.plot()	df['Value'].plot()

Comparison of Pandas and NumPy

Feature	Pandas	NumPy
Data Type	Heterogeneous	Homogeneous (numerical data)
Data Structure	DataFrame (rows and columns)	Array (multi-dimensional)
Use Case	Data manipulation, cleaning	Mathematical computations
Best For	Tabular data, mixed data types	Numerical tasks, matrix ops
Performance	Slower than NumPy for large datasets	Optimized for large arrays

---

Conclusion

Both NumPy and Pandas are essential libraries in Python for data analysis and scientific computing, each serving a distinct yet complementary purpose. NumPy excels in handling numerical computations, matrix operations, and high-performance tasks, making it ideal for working with large datasets and performing complex mathematical functions. On the other hand, Pandas offers robust tools for data manipulation, cleaning, and analysis, particularly with structured data like tabular formats. Its versatility in handling heterogeneous data types, missing values, and easy integration with other libraries makes it indispensable for data analysis tasks. Together, NumPy and Pandas form the backbone of data science in Python, equipping professionals with the necessary tools to efficiently handle and analyze data in various fields.