UNIT-2

1 What is NumPy, and why is it important in Python for numerical computations ?.

NumPy (**Numerical Python**) is a **Python library** used for **fast numerical computations**.
It provides an efficient way to store and manipulate **large multi-dimensional arrays** and offers a wide range of **mathematical functions** to operate on them.

Why NumPy is Important

1. **Speed** – Operations are implemented in optimized C code, making them much faster than Python loops.
2. **Memory Efficiency** – Stores data in contiguous memory blocks, unlike Python lists.
3. **Vectorization** – Allows operations on entire arrays without writing explicit loops.
4. **Mathematical Power** – Supports linear algebra, Fourier transforms, statistics, and more.
5. **Foundation for Data Science** – Libraries like Pandas, SciPy, scikit-learn, and TensorFlow are built on top of NumPy.

Example

python Copy Edit

```python
import numpy as np

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

# Element-wise addition
print(a + b)    # [5 7 9]

# Mean
print(np.mean(a))  # 2.0
```

Here, a + b is computed **without loops**, making it **faster and cleaner**.

Key Features of NumPy

1. **N-Dimensional Array Object (`ndarray`)**
   - Core data structure for storing and manipulating large datasets efficiently.
   - Much faster than Python's built-in lists for numerical operations.
2. **Vectorized Operations**
   - Eliminates the need for explicit loops; operations are applied element-wise to entire arrays at once.
3. **Mathematical & Statistical Functions**
   - Functions like `sum()`, `mean()`, `std()`, `dot()`, `sin()`, `exp()` are built in and optimized.
4. **Linear Algebra Support**
   - Matrix multiplication, determinants, eigenvalues, etc.
5. **Random Number Generation**
   - Useful for simulations, machine learning, and testing.
6. **Interoperability**
   - Works seamlessly with libraries like **Pandas**, **Matplotlib**, **SciPy**, and **TensorFlow**.

2.How do you create a NumPy array? Mention any two attributes of a NumPy array and explain their significance.

1. Creating a NumPy Array

You can create a NumPy array using the `numpy.array()` function by passing in a Python list (or other sequence):

```python
import numpy as np

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

This creates a 1-dimensional array containing the integers 1 to 5.

You can also create multi-dimensional arrays or use built-in functions like `np.zeros()`, `np.ones()`, and `np.arange()`:

```python
arr2d = np.array([[1, 2, 3], [4, 5, 6]])  # 2D array
```

2. Attributes of a NumPy Array and Their Significance

a) `shape`

- **Definition:**

  The `shape` attribute is a tuple indicating the number of elements in each dimension (rows, columns, etc.) of the array.

- **Significance:**

  It helps you understand the arrangement and organization of your data. For example:

  ```python
  arr2d.shape   # Output: (2, 3)  -> 2 rows, 3 columns
  ```

  Knowing the shape is essential for reshaping, broadcasting, and performing mathematical operations safely.

b) `dtype`

- **Definition:**

  The `dtype` attribute describes the type of elements stored in the array (e.g., `int32`, `float64`, etc.).

- **Significance:**

  It informs you of the kind of data your array holds and how much memory each item consumes.
  Example:

  ```python
  arr.dtype   # Output: dtype('int64') (on a 64-bit system)
  ```

  Choosing the right `dtype` ensures correct calculations and efficient memory usage.

3.What are the two main data structures in Pandas? Briefly explain each with an example.

In Pandas, the two main data structures are:

---

1. Series

- **Definition:** A **1-dimensional** labeled array that can hold any data type (integers, floats, strings, objects, etc.).
- **Labels (Index):** Every element in a Series has an **index** (like row labels) for easy data access.
- **Think of it like:** A column in Excel or a single column of a DataFrame.

Example:

python Copy Edit

```python
import pandas as pd

# Creating a Series from a list
s = pd.Series([10, 20, 30, 40], index=['a', 'b', 'c', 'd'])
print(s)
```

Output:

css Copy Edit

```css
a    10
b    20
c    30
d    40
dtype: int64
```

Here:

- Values → `[10, 20, 30, 40]`
- Index → `['a', 'b', 'c', 'd']`

2. DataFrame

- **Definition:** A **2-dimensional** labeled data structure with rows and columns.
- **Think of it like:** A table in Excel or SQL — each column is a Series.
- **Features:** Can store different data types in each column.

Example:

python Copy Edit

```python
data = {
    'Name': ['Alice', 'Bob', 'Charlie'],
    'Age': [25, 30, 35],
    'City': ['New York', 'London', 'Paris']
}

df = pd.DataFrame(data)
print(df)
```

Output:

pgsql Copy Edit

```pgsql
      Name  Age      City
0    Alice   25  New York
1      Bob   30    London
2  Charlie   35     Paris
```

Here:

- Rows have **row indexes** (`0, 1, 2` by default).
- Columns have **column labels** (`'Name'`, `'Age'`, `'City'`).

4.What is a scatter plot? Write a short example using plt.scatter() to visualize a dataset.

Scatter Plot

A **scatter plot** is a type of plot that displays data points on a **two-dimensional coordinate system**, where:

- Each point represents the relationship between two variables (x and y).
- It's useful for identifying **patterns**, **correlations**, or **outliers** in data.

Example using `plt.scatter()`

python Copy Edit

```python
import matplotlib.pyplot as plt

# Sample dataset
x = [5, 7, 8, 7, 6, 9, 5, 8, 7, 6]
y = [99, 86, 87, 88, 100, 86, 103, 87, 94, 78]

# Create scatter plot
plt.scatter(x, y, color='blue', marker='o')

# Add labels and title
plt.xlabel("X values")
plt.ylabel("Y values")
plt.title("Sample Scatter Plot")

# Show plot
plt.show()
```

Explanation:

- `x` → values on the horizontal axis.
- `y` → values on the vertical axis.
- `color` and `marker` → optional styling parameters.

5.How does NumPy sort arrays? Differentiate between np.sort() and ndarray.sort() with one example.

NumPy Sorting Overview

NumPy sorts arrays using an **efficient sorting algorithm** (by default **quicksort**, but you can choose `'mergesort'`, `'heapsort'`, or `'stable'`).
Sorting can be **along a specific axis** or the **entire flattened array**.

Difference Between `np.sort()` and `ndarray.sort()`

Feature	`np.sort()`	`ndarray.sort()`
Type	Function in the NumPy module	Method of a NumPy array object
Returns	**New sorted array** (original array unchanged)	Sorts **in place** (modifies the original array)
Usage	`np.sort(array, axis=-1)`	`array.sort(axis=-1)`
Mutability	Does not alter the original array	Alters the original array

Example

python

Copy

```python
import numpy as np

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

# Using np.sort() - returns a sorted copy
sorted_arr = np.sort(arr)
print("np.sort() result:", sorted_arr)    # [1 2 3]
print("Original array after np.sort():", arr)  # [3 1 2]

# Using ndarray.sort() - sorts in place
arr.sort()
print("Array after ndarray.sort():", arr)  # [1 2 3]
```

Key takeaway:

- Use `np.sort()` if you **need the original array unchanged.**
- Use `ndarray.sort()` if you **want to modify the array directly** (more memory-efficient).

6.Explain how aggregation functions like min(), max(), mean() and Aggregating Along an Axis are used in NumPy. Provide one code example.

Aggregation Functions in NumPy

Aggregation functions in NumPy perform calculations that **summarize data** from an array, such as finding the minimum, maximum, or average.

Common ones include:

- `min()` → smallest element
- `max()` → largest element
- `mean()` → average value

Aggregating Along an Axis

- `axis=None` (default) → operation is applied to the **entire array** (all elements).
- `axis=0` → operation is applied **column-wise** (down each column).
- `axis=1` → operation is applied **row-wise** (across each row).

Example

python

Copy Edit

```python
import numpy as np

# 2D array
arr = np.array([[10, 20, 30],
                [40, 50, 60]])

# Aggregating without axis (entire array)
print("Min:", np.min(arr))     # 10
print("Max:", np.max(arr))     # 60
print("Mean:", np.mean(arr))   # 35.0

# Aggregating along axis 0 (column-wise)
print("Column-wise min:", np.min(arr, axis=0))  # [10 20 30]
print("Column-wise max:", np.max(arr, axis=0))  # [40 50 60]
print("Column-wise mean:", np.mean(arr, axis=0))# [25. 35. 45.]

# Aggregating along axis 1 (row-wise)
print("Row-wise min:", np.min(arr, axis=1))     # [10 40]
print("Row-wise max:", np.max(arr, axis=1))     # [30 60]
print("Row-wise mean:", np.mean(arr, axis=1))   # [20. 50.]
```

7.Write a Python program using Matplotlib to create a line plot for a dataset. Add labels, a title, and a legend. Explain each component used in the plot.

Python Program

python

Copy Edit

```python
import matplotlib.pyplot as plt

# Sample dataset
x = [1, 2, 3, 4, 5]        # X-axis values
y1 = [2, 4, 6, 8, 10]      # Y-axis values for first line
y2 = [1, 3, 5, 7, 9]       # Y-axis values for second line

# Create line plots
plt.plot(x, y1, label='Line 1', color='blue', marker='o')
plt.plot(x, y2, label='Line 2', color='red', linestyle='--', marker='x')

# Add labels for axes
plt.xlabel("X-axis Label")
plt.ylabel("Y-axis Label")

# Add title
plt.title("Sample Line Plot")

# Add legend
plt.legend()

# Display plot
plt.show()
```

Explanation of Components

1. **`import matplotlib.pyplot as plt`**
   - Imports Matplotlib's plotting module and gives it the alias `plt`.
2. **`x, y1, y2`**
   - Lists containing the dataset for the x-axis and y-axis values.
3. **`plt.plot()`**
   - Plots the data as a line graph.
   - `label` → name shown in the legend.
   - `color` → sets the line color.
   - `marker` → marks data points (`'o'`, `'x'`, `'^'`, etc.).
   - `linestyle` → style of the line (`'-'` solid, `'--'` dashed).
4. **`plt.xlabel()` and `plt.ylabel()`**
   - Adds descriptive labels to the X-axis and Y-axis.
5. **`plt.title()`**
   - Adds a title to the entire plot.
6. **`plt.legend()`**
   - Displays a legend to differentiate multiple plotted lines using their `label`.
7. **`plt.show()`**
   - Renders and displays the plot.

8.What functions does Pandas provide to detect and handle missing data? Mention two methods for dealing with missing values.

Detecting Missing Data in Pandas

Pandas provides functions to **detect** and **handle** missing data (`NaN` values):

Detection Functions

1. `isnull()` → Returns a boolean mask indicating `True` where values are missing.
2. `notnull()` → Returns `True` where values are **not** missing.

Example:

python Copy Edit

```python
import pandas as pd
import numpy as np

df = pd.DataFrame({
    'A': [1, np.nan, 3],
    'B': [4, 5, np.nan]
})

print(df.isnull())    # True where NaN exists
print(df.notnull())  # True where value is present
```

Two Methods to Handle Missing Values

1. `dropna()` – Remove missing values (rows or columns).

python Copy Edit

```python
df_dropped = df.dropna()        # Drop rows with NaN
df_dropped_cols = df.dropna(axis=1)  # Drop columns with NaN
```

2. `fillna()` – Replace missing values with a specified value or method.

python Copy Edit

```python
df_filled = df.fillna(0)             # Replace NaN with 0
df_forward = df.fillna(method='ffill')  # Forward fill (use previous value)
```

✅ Summary Table:

Purpose	Function
Detect missing	`isnull()`, `notnull()`
Remove missing	`dropna()`
Replace missing	`fillna()`

9.Explain the structure and key attributes of NumPy arrays. How can arrays be created using functions like np.array(), np.zeros(), and np.arange()? Provide examples and explain each.

Structure of a NumPy Array

A **NumPy array** (technically `numpy.ndarray`) is:

- **N-dimensional**: Can be 1D (vector), 2D (matrix), or higher.
- **Homogeneous**: All elements have the same **data type** (`dtype`).
- **Contiguous in memory**: Enables **fast vectorized operations**.
- Has **axes** (dimensions) with lengths stored in its **shape**.

Key Attributes of NumPy Arrays

Attribute	Description	Example
`ndim`	Number of dimensions (axes).	`2` for a 2D matrix
`shape`	Tuple of array dimensions.	`(3, 4)` means 3 rows × 4 columns
`size`	Total number of elements.	`12` for `(3, 4)`
`dtype`	Data type of elements.	`int32`, `float64`, etc.
`itemsize`	Bytes per element.	`8` for `float64`
`nbytes`	Total bytes consumed.	`size * itemsize`

Creating Arrays

1. Using `np.array()`

Converts Python lists (or nested lists) into NumPy arrays.

python Copy Edit

```python
import numpy as np

arr = np.array([1, 2, 3, 4, 5])
print(arr)
print("Shape:", arr.shape)   # (5,)
print("ndim:", arr.ndim)     # 1
print("dtype:", arr.dtype)   # int64 (may vary by system)
```

✅ **Use when:** You have existing Python lists or tuples you want to convert.

2. Using `np.zeros()`

Creates an array filled with zeros of a given shape.

python Copy Edit

```python
zeros_arr = np.zeros((2, 3))
print(zeros_arr)
print("Shape:", zeros_arr.shape)  # (2, 3)
```

✅ **Use when:** You need a placeholder array to fill later (e.g., for algorithms).

3. Using `np.arange()`

Creates evenly spaced values within a given range (like Python's `range()`, but returns an array).

python Copy Edit

```python
range_arr = np.arange(0, 10, 2)  # Start=0, Stop=10, Step=2
print(range_arr)
```

✅ **Use when:** You need sequences for indexing, plotting, or numeric ranges.

Extra Examples

python Copy Edit

```python
# Array of ones
ones_arr = np.ones((3, 2))

# Array with evenly spaced values
linspace_arr = np.linspace(0, 1, 5)  # 5 numbers between 0 and 1
```

Summary Table

Function	Purpose	Example Output
`np.array()`	Convert list/tuple to array	`[1 2 3]`
`np.zeros()`	Create zero-filled array	`[[0. 0.], [0. 0.]]`
`np.arange()`	Create range of values	`[0 2 4 6 8]`
`np.ones()`	Create one-filled array	`[[1. 1.], [1. 1.]]`
`np.linspace()`	Evenly spaced numbers	`[0. 0.25 0.5 0.75 1.]`

10.Describe Series and DataFrame objects in Pandas. What are their key characteristics? How are they created? Compare and contrast Series and DataFrame with relevant code examples

1. Series in Pandas

Definition

A **Series** is a **1-dimensional labeled array** that can hold any data type — integers, floats, strings, Python objects, etc.
Think of it like **a single column** in an Excel sheet.

Key Characteristics

- **1D data** with labels (**index**).
- Can hold **different data types** (but usually consistent).
- Index labels allow fast lookups.
- Built on top of NumPy arrays.

Creating a Series

python

```python
import pandas as pd

# Creating from a list
s = pd.Series([10, 20, 30, 40], index=['a', 'b', 'c', 'd'])
print(s)
```

Output:

css

```css
a    10
b    20
c    30
d    40
dtype: int64
```

---

2. DataFrame in Pandas

Definition

A **DataFrame** is a **2-dimensional labeled data structure** with rows and columns — essentially a **table**.
Think of it like an **Excel spreadsheet** or a **SQL table**.

Key Characteristics

- **2D data** with row and column labels.
- Columns can hold **different data types**.
- Each column is essentially a Pandas **Series**.
- Flexible for data manipulation and analysis.

Creating a DataFrame

python

```python
# Creating from a dictionary
data = {
    'Name': ['Alice', 'Bob', 'Charlie'],
    'Age': [25, 30, 35],
    'City': ['New York', 'London', 'Paris']
}

df = pd.DataFrame(data)
print(df)
```

Output:

pgsql

```
      Name  Age      City
0    Alice   25  New York
1      Bob   30    London
2  Charlie   35     Paris
```

3. Series vs DataFrame — Comparison

Feature	Series	DataFrame
Dimensions	1D	2D
Structure	Single column	Multiple columns
Index	Single index (row labels)	Row index + column labels
Data Types	Homogeneous (one type per Series)	Heterogeneous (each column can be a different type)
Analogy	Column in Excel	Table in Excel
Created from	List, array, dict, scalar	Dict of lists, 2D array, list of dicts

4. Code Example — Side-by-Side

python

```python
# Series example
s = pd.Series([100, 200, 300], index=['X', 'Y', 'Z'])
print("Series:\n", s)

# DataFrame example
df = pd.DataFrame({
    'Product': ['A', 'B', 'C'],
    'Price': [100, 200, 300]
})
print("\nDataFrame:\n", df)
```

Output:

makefile

```
Series:
X    100
Y    200
Z    300
dtype: int64

DataFrame:
  Product  Price
0       A    100
1       B    200
2       C    300
```

11.How do you save a plot using savefig() in Matplotlib? Discuss the parameters such as filename, dpi, and format. Provide examples of saving a plot as PNG and PDF.

`savefig()` in Matplotlib

The `savefig()` function saves the current figure to a file.

Syntax

python Copy Edit

```python
plt.savefig(fname, dpi=None, format=None, ...)
```

Key Parameters

1. **`fname` (filename)**
   - The file name (with path if needed).
   - Example: `"plot.png"`, `"C:/plots/graph.pdf"`.
2. **`dpi` (dots per inch)**
   - Controls the **resolution** of the saved image.
   - Higher `dpi` → better quality but larger file size.
   - Example: `dpi=300` (good for printing).
3. **`format`**
   - File format (`'png'`, `'pdf'`, `'jpg'`, `'svg'`, etc.).
   - If not provided, Matplotlib infers from the file extension.
4. **Other useful parameters**
   - `bbox_inches='tight'` → trims extra whitespace.
   - `transparent=True` → saves with a transparent background.

Example — Saving as PNG

python Copy Edit

```python
import matplotlib.pyplot as plt

# Sample plot
x = [1, 2, 3, 4, 5]
y = [2, 4, 6, 8, 10]

plt.plot(x, y, label='Line', color='blue', marker='o')
plt.xlabel("X-axis")
plt.ylabel("Y-axis")
plt.title("Sample Plot")
plt.legend()

# Save as PNG
plt.savefig("plot.png", dpi=300, format='png', bbox_inches='tight')
plt.show()
```

Example — Saving as PDF

python

```python
plt.plot(x, y, color='red', marker='x')
plt.title("PDF Example")

# Save as PDF
plt.savefig("plot.pdf", dpi=300, format='pdf', bbox_inches='tight')
plt.show()
```

12.What are the key components of a plot in Matplotlib? Discuss general tips for making effective visualizations such as using titles, labels, legends, and styles. Provide examples.

1. Key Components of a Matplotlib Plot

When you create a plot in Matplotlib, it usually consists of these core parts:

Component	Description
Figure	The entire plotting area (can contain multiple subplots).
Axes	The actual area where data is plotted (inside the figure).
Title	Describes what the plot is about.
Axis Labels	Names for the X and Y axes to clarify meaning.
Ticks & Tick Labels	Marks along the axes showing measurement intervals.
Legend	Explains what each line, color, or marker represents.
Grid	Helps visually align data points.
Style	The overall look (colors, line styles, background themes).

2. General Tips for Effective Visualizations

- **Use a descriptive title** → helps the viewer instantly understand the purpose of the plot.
- **Label axes clearly** → tell exactly what the X and Y values represent.
- **Include a legend** if there's more than one data series.
- **Choose colors and markers wisely** → avoid too many, keep contrast high.
- **Use grid lines** for easier value reading.
- **Maintain aspect ratio and figure size** for clarity.
- **Use consistent style** if multiple plots are shown together.

3. Example — A Well-Formatted Plot

python

```python
import matplotlib.pyplot as plt

# Sample data
x = [1, 2, 3, 4, 5]
y1 = [2, 4, 6, 8, 10]
y2 = [1, 4, 9, 16, 25]

# Set figure size for better visibility
plt.figure(figsize=(8, 5))

# Plot multiple lines with labels
plt.plot(x, y1, label='Linear Growth', color='blue', marker='o', linestyle='-')
plt.plot(x, y2, label='Quadratic Growth', color='red', marker='s', linestyle='--')

# Add title and labels
plt.title("Linear vs Quadratic Growth", fontsize=16, fontweight='bold')
plt.xlabel("X Values", fontsize=12)
plt.ylabel("Y Values", fontsize=12)

# Add grid
plt.grid(True, linestyle=':', alpha=0.7)

# Add legend
plt.legend()

# Apply a style
plt.style.use('seaborn-v0_8')

plt.show()
```

Output Explanation

- **Title:** `"Linear vs Quadratic Growth"` — tells us what we're looking at.
- **X-axis label:** `"X Values"` — explains what the horizontal values mean.
- **Y-axis label:** `"Y Values"` — explains what the vertical values mean.
- **Legend:** Helps distinguish between the **blue linear line** and the **red quadratic curve**.
- **Grid:** Dotted lines make it easier to read values.
- **Style (** `seaborn` **):** Gives a cleaner look with a light background.

✅ **Quick Style Tip:**

Matplotlib supports many built-in styles:

```python
plt.style.available
```

Try:

```python
plt.style.use('ggplot')  # Clean red-grid style
```

or

```python
plt.style.use('fivethirtyeight')  # Popular blog-style charts
```