THEORY QUESTIONS

Q1. What is NumPy, and why is it widely used in Python?

Explanation:

NumPy (Numerical Python) is a powerful open-source library in Python that provides support for numerical computations, linear algebra, and handling large multi-dimensional arrays and matrices efficiently.

Key Features of NumPy:

1.N-dimensional array object (ndarray)

- More powerful and memory-efficient than Python's built-in lists.
- Allows element-wise operations without explicit loops.

2.Mathematical functions

- Provides fast vectorized operations (addition, subtraction, multiplication, division).
- Includes advanced functions for linear algebra, statistics, and Fourier transforms.

3.Broadcasting

- Automatically expands arrays of different shapes to perform element-wise operations.

4.Integration

- Works well with other Python libraries like Pandas, Matplotlib, TensorFlow, and Scikit-learn.
- Often serves as the foundation for scientific computing in Python.

5.Performance

- Written in C and optimized for performance, making numerical operations significantly faster than using plain Python lists.

Why NumPy is Widely Used:

- Speed: Vectorized operations make it much faster than standard Python loops.
- Simplicity: Makes numerical code easier to write and understand.
- Foundation for Data Science & AI: Most data science, machine learning, and scientific libraries are built on top of NumPy.
- Community Support: Large user base, extensive documentation, and wide adoption in academia and industry.

✅ Example:

```
import numpy as np

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

# Vectorized operations
print(a + b)    # [5 7 9]
print(a * b)    # [ 4 10 18]

# 2D array
matrix = np.array([[1, 2], [3, 4]])
print(matrix.T)  # Transpose
```

Q2. How does broadcasting work in NumPy?

Explanation:

Broadcasting in NumPy means performing operations on arrays of different shapes without making explicit copies. If one array has a smaller dimension, NumPy automatically stretches it to match the larger one (without using extra memory).

Example 1: Scalar with Array

```
import numpy as np

a = np.array([1, 2, 3])
print(a + 5)
```

Output:

```
 [6 7 8]
```

Here, 5 is broadcast to [5, 5, 5] and added to each element.

Example 2: 2D with 1D

```
A = np.array([[1, 2, 3],
              [4, 5, 6]])
b = np.array([10, 20, 30])

print(A + b)
```

Output:

```
[[11 22 33]
[14 25 36]]
```

👉 b (shape (3,)) is broadcast to match A (shape (2,3)), and addition happens element-wise.

Q3. What is a Pandas DataFrame?

Explanation:

A Pandas DataFrame is a two-dimensional, tabular data structure in Python provided by the Pandas library. It looks and behaves a lot like an Excel spreadsheet or SQL table, with rows and columns.

Key Features of a DataFrame:

1.Labeled axes → both rows and columns have labels (row index, column names).

2.Heterogeneous data → each column can store a different data type (integers, floats, strings, etc.).

3.Size mutability → you can insert, delete, or modify rows and columns easily.

4.Powerful operations → filtering, grouping, merging, handling missing data, and statistical analysis.

Example:

import pandas as pd

```
# Creating a DataFrame from a dictionary
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
```

✅ In short:

A DataFrame is the most commonly used Pandas object for data analysis — it's basically a flexible table that makes it easy to clean, analyze, and visualize data.

Q4. Explain the use of the groupby() method in Pandas.

Explanation:

The groupby() method in Pandas is used to split data into groups based on one or more column values, so you can apply operations like aggregation, transformation, or filtering on each group separately.

It follows the Split → Apply → Combine strategy:

1.Split – Divide the DataFrame into groups based on column(s).

2.Apply – Apply a function (like sum(), mean(), count(), etc.) to each group.

3.Combine – Merge the results back into a DataFrame or Series.

🔹 Example 1: Basic Grouping

```
import pandas as pd

data = {
"Department": ["IT", "HR", "IT", "Finance", "HR", "Finance"],
"Employee": ["A", "B", "C", "D", "E", "F"],
"Salary": [50000, 40000, 60000, 55000, 42000, 58000]
}
```

```
df = pd.DataFrame(data)

# Group by department and calculate average salary
grouped = df.groupby("Department")["Salary"].mean()
print(grouped)
```

Output:

```
Department
Finance    56500.0
HR         41000.0
IT         55000.0
Name: Salary, dtype: float64
```

🔹 Example 2: Multiple Aggregations

```
# Multiple aggregations on Salary
result = df.groupby("Department")["Salary"].agg(["mean", "max", "min"])
print(result)
```

Output:

```
              mean    max    min
Department
Finance    56500.0  58000  55000
HR         41000.0  42000  40000
IT         55000.0  60000  50000
```

✅ In short:

groupby() is used in Pandas to analyze data by groups, making it easy to compute summary statistics (like sum, mean, count) or apply custom functions.

Q5. Why is Seaborn preferred for statistical visualizations?

Explanation:

Seaborn is often preferred for statistical visualizations in Python because it is built on top of Matplotlib and integrates tightly with Pandas, making it easier to create beautiful, informative, and statistically meaningful plots with very little code.

🔹 Why Seaborn is Preferred:

1.High-level interface

- Requires fewer lines of code compared to Matplotlib for complex plots.
- Automatically handles figure styling, color palettes, and legends.

2.Statistical functions built-in

- Supports plots that show relationships and distributions, such as:
  - distplot / histplot (distribution of data)
  - boxplot & violinplot (data spread & outliers)
  - regplot / lmplot (regression with confidence intervals)
  - heatmap (correlation matrix visualization).

3.Integration with Pandas

- Works directly with DataFrames, so you can pass column names instead of manually extracting arrays.

4.Attractive default styles

- Provides modern, publication-quality themes without needing extra customization.

5.Supports statistical summaries

- Automatically adds confidence intervals, trend lines, and aggregation in many plots.

🔹 **Example**

```
import seaborn as sns
import pandas as pd

# Sample dataset
tips = sns.load_dataset("tips")

# Scatterplot with regression line
sns.lmplot(x="total_bill", y="tip", data=tips)
```

This creates a scatter plot with a best-fit regression line + confidence interval — something that would require much more code in Matplotlib.

✅ In short:

Seaborn is preferred for statistical visualizations because it makes it easy, quick, and visually appealing to explore and communicate data distributions, relationships, and trends.

Q6. What are the differences between NumPy arrays and Python lists?

Explanation:

Python lists and NumPy arrays often look similar, but they are quite different in how they work and what they're used for.

🔹 Key Differences Between Python Lists and NumPy Arrays

Feature	Python List	NumPy Array
Type of elements	Can store different data types (int, float, string, etc. in the same list)	Stores elements of the **same data type** (homogeneous)
Memory efficiency	Takes more memory, elements are stored as Python objects	More memory-efficient, stored in contiguous blocks (C-style)
Speed	Slower for numerical operations (requires loops)	Much faster (vectorized operations, implemented in C)

Feature	Python List	NumPy Array
Operations	Supports basic operations (append, insert, slicing, etc.)	Supports advanced mathematical operations (matrix multiplication, linear a
Dimension support	Mostly 1D (lists of lists for higher dimensions, but messy)	Supports **N-dimensional arrays** (1D, 2D, 3D, ...)
Best suited for	General-purpose collections of objects	Numerical and scientific computing

🔹 Example

Python List

```
lst = [1, 2, 3]
print(lst * 2)   # [1, 2, 3, 1, 2, 3]  (repeats list)
```

NumPy Array

```
import numpy as np

arr = np.array([1, 2, 3])
print(arr * 2)   # [2 4 6]  (element-wise multiplication)
```

👉 Notice how the same operation behaves differently!

✅ In short:

- Use Python lists for general-purpose storage.
- Use NumPy arrays when working with large datasets, mathematical operations, or scientific computing because they are faster and more efficient.

Q7. What is a heatmap, and when should it be used?

Explanation:

A heatmap is a type of data visualization that uses color intensity to represent values in a matrix or 2D dataset. Instead of showing numbers directly, the values are mapped to colors (e.g., darker shades for higher values, lighter shades for lower values).

🔹 When to Use a Heatmap

1.Correlation analysis

- To quickly see how variables are related (e.g., in a correlation matrix).

2.Pattern detection

- Helps spot clusters, trends, or anomalies in large datasets.

3.Comparing values across categories

- Useful for categorical vs. numerical data (e.g., average sales per region per month).

4.Visualizing large matrices

- Ideal for representing large 2D arrays, confusion matrices, or similarity matrices.

🔹 Example in Python (Seaborn Heatmap)

```
import seaborn as sns
import pandas as pd

# Sample data: correlation matrix
data = {
"Math": [85, 90, 78, 92, 88],
"Science": [80, 85, 75, 95, 89],
"English": [78, 82, 88, 90, 85]
}
```

```
df = pd.DataFrame(data)

# Create heatmap of correlations
sns.heatmap(df.corr(), annot=True, cmap="coolwarm")
```

This will show a correlation heatmap where:

- 1.0 (red) means perfect positive correlation.
- -1.0 (blue) means perfect negative correlation.
- 0 means no correlation.

✅ In short:

A heatmap is best used when you want to visualize relationships, trends, or patterns in tabular data, especially for correlations and large matrices.

Q8. What does the term “vectorized operation” mean in NumPy?

Explanation:

In NumPy, a vectorized operation means performing an operation on entire arrays (vectors, matrices) at once, instead of looping through individual elements with for loops.

These operations are implemented in optimized C code under the hood, so they run much faster than pure Python loops.

🔹 Key Points

- Works on arrays element-wise.
- Eliminates the need for explicit Python loops.
- Much faster and more concise.
- Makes code easier to read.

🔹 Example: Without vs With Vectorization

Without vectorization (using loops):

```
import numpy as np

a = [1, 2, 3, 4]
b = [5, 6, 7, 8]

result = []
for i in range(len(a)):
result.append(a[i] + b[i])

print(result)   # [6, 8, 10, 12]
```

With NumPy vectorization:

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

result = a + b
print(result)   # [ 6  8 10 12 ]
```

👉 The second approach is shorter, cleaner, and faster.

✅ In short:

A vectorized operation in NumPy means applying a function or operator directly to a whole array, letting NumPy handle the looping internally in optimized C code — giving you speed and simplicity.

Q9. How does Matplotlib differ from Plotly?

Explanation:

both Matplotlib and Plotly are popular Python libraries for data visualization, but they serve different purposes and have distinct strengths.

🔹 Key Differences Between Matplotlib and Plotly

Feature	Matplotlib	Plotly
Type	Static plotting library (can make interactive with add-ons)	Interactive plotting library (built-in interactivity)
Ease of use	Low-level → more code needed for customization	High-level → easier to create interactive, polished plots
Output	Mainly static images (PNG, PDF, etc.), but can embed in Jupyter	Interactive plots (zoom, pan, hover tooltips) in browsers and notebooks
Best for	Scientific research, static reports, publications	Dashboards, web apps, and interactive data exploration
Customization	Very flexible, but requires more coding	Easier styling and interactivity with less effort
Integration	Works well with Seaborn, Pandas, NumPy	Works well with Dash (for web dashboards) and Pandas
Community	Older, larger community, widely used in academia	Growing community, widely used in data science dashboards and business apps

🔹 Example Comparison

Matplotlib (static plot):

```
import matplotlib.pyplot as plt

x = [1, 2, 3, 4, 5]
y = [10, 20, 25, 30, 40]

plt.plot(x, y, marker='o')
plt.title("Matplotlib Example")
plt.xlabel("X-axis")
plt.ylabel("Y-axis")
plt.show()
```

Plotly (interactive plot):

```
import plotly.express as px

x = [1, 2, 3, 4, 5]
y = [10, 20, 25, 30, 40]

fig = px.line(x=x, y=y, markers=True, title="Plotly Example")
fig.show()
```

👉 The Matplotlib plot is static, while the Plotly plot lets you zoom, hover, and interact directly in your browser or notebook.

✅ In short:

- Use Matplotlib when you need static, publication-quality plots or full control over plot details.
- Use Plotly when you need interactive, web-friendly visualizations or are building dashboards.

Q10. What is the significance of hierarchical indexing in Pandas?

Explanation:

hierarchical indexing (also called a MultiIndex) in Pandas is a powerful feature that lets you have multiple levels of indexing (row or column labels) within a DataFrame or Series.

🔹 Significance of Hierarchical Indexing

1.Work with higher-dimensional data in 2D form

- Lets you represent multi-dimensional data (like 3D or 4D) in a 2D DataFrame.

2.More flexible data organization

- Rows and columns can be grouped logically using multiple keys.

3.Easier grouping and aggregation

- Useful for operations like groupby(), pivot tables, and reshaping (stack / unstack).

4.Better slicing and subsetting

- You can access data at multiple levels of granularity (e.g., get all data for a particular year, then filter by month).

5.Cleaner representation

- Makes complex datasets easier to handle without resorting to deeply nested dictionaries or reshaping arrays.

🔹 **Example: MultiIndex in Pandas**

```
import pandas as pd

# Create sample data
data = {
"Sales": [200, 220, 250, 270],
"Profit": [50, 60, 65, 80]
 }

index = pd.MultiIndex.from_tuples(
[("2023", "Q1"), ("2023", "Q2"), ("2024", "Q1"), ("2024", "Q2")],
names=["Year", "Quarter"]
)

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

Output:

```
        Sales  Profit
     Year Quarter
2023 Q1       200      50
     Q2       220      60
```

```
2024 Q1       250      65
     Q2       270      80
```

👉 Here, rows are indexed by both Year and Quarter, allowing queries like:

```
print(df.loc["2023"])      # Get all 2023 data
print(df.loc[("2024", "Q1")])  # Get 2024 Q1 data
```

✅ In short:

Hierarchical indexing in Pandas allows you to work with multi-dimensional, structured data more effectively, making it easier to organize, slice, and analyze complex datasets.

Q11. What is the role of Seaborn's pairplot() function?

Explanation:

The pairplot() function in Seaborn is used to create a grid of plots that shows relationships between pairs of variables in a dataset, along with distributions of individual variables.

It's especially useful for exploratory data analysis (EDA) when you want to quickly understand how multiple variables relate to each other.

🔹 **Role of pairplot()**

1.Visualize pairwise relationships

- Creates scatterplots for every possible pair of numeric columns.

2.Show distributions

- Displays histograms (or KDE plots) along the diagonal to show variable distributions.

3.Identify correlations and trends

- Makes it easier to spot linear relationships, clusters, or outliers.

4.Group comparisons

- You can color-code points (hue parameter) to compare categories.

🔹 **Example**

```
import seaborn as sns

# Load sample dataset
iris = sns.load_dataset("iris")

# Create pairplot
sns.pairplot(iris, hue="species")
```

✅ **This will:**

- Plot scatterplots of all combinations of features (sepal_length, sepal_width, petal_length, petal_width).
- Show histograms along the diagonal.
- Use different colors for each species of iris.

🔹 **When to Use**

- During exploratory data analysis to get an overview of dataset structure.
- To detect relationships, clusters, or separation between categories.
- To check for linear separability before applying models like classification.

✅ In short:

pairplot() in Seaborn is a quick way to visualize all pairwise relationships and distributions in a dataset, making it extremely useful for multivariate data exploration.

Q12. What is the purpose of the describe() function in Pandas?

Explanation:

The describe() function in Pandas is used to generate summary statistics of a DataFrame or Series. It provides a quick overview of the distribution, central tendency, and spread of the data.

- **Purpose of describe()**

1.Quick summary ⟶ Gives key statistics without writing multiple functions.

2.Data understanding ⟶ Helps in exploratory data analysis (EDA).

3.Detect issues ⟶ Identify missing values, outliers, or unexpected ranges.

4.Compare features ⟶ Understand how numerical columns differ in scale and distribution.

- **Example**

```
import pandas as pd

data = {
    "Age": [22, 25, 30, 28, 35],
    "Salary": [40000, 50000, 60000, 55000, 70000]
}

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

Output:

```
             Age        Salary
count    5.00000      5.000000
mean    28.00000  55000.000000
std      4.69042  11180.339887
min     22.00000  40000.000000
25%     25.00000  50000.000000
50%     28.00000  55000.000000
```

```
75%     30.00000  60000.000000
max     35.00000  70000.000000
```

🔹 What it shows (for numeric data)

- count ⟶ number of non-missing values
- mean ⟶ average value
- std ⟶ standard deviation (spread)
- min / max ⟶ minimum and maximum values
- 25%, 50%, 75% ⟶ quartiles (useful for spotting skewness/outliers)

👉 For categorical data, it instead shows:

- count, unique, top, freq

✅ In short:

The describe() function in Pandas provides a statistical summary of your dataset, making it one of the most useful tools for exploratory data analysis.

Q13. Why is handling missing data important in Pandas?

Explanation:

Handling missing data in Pandas is important because real-world datasets are often incomplete, and ignoring missing values can lead to wrong conclusions, errors in analysis, or broken models.

🔹 Why It's Important

1.Accuracy of analysis

- Missing values can skew statistics like mean, median, or correlation.
- Example: If some salaries are missing, the average salary may be misleading.

2.Compatibility with functions & models

- Many machine learning algorithms and visualization tools cannot handle NaN values.
- If not handled, they may throw errors or drop large chunks of data automatically.

3.Data integrity

- Filling or removing missing values ensures a clean, consistent dataset.

4.Better decision-making

- Different strategies (like imputation, dropping, or flagging) can reduce bias and improve insights.

🔹 **Example in Pandas**

```
import pandas as pd
import numpy as np

data = {
    "Name": ["Alice", "Bob", "Charlie", "David"],
    "Age": [25, np.nan, 30, 28],
    "Salary": [50000, 60000, np.nan, 55000]
}

df = pd.DataFrame(data)
print("Original Data:\n", df)

# Handling missing values
df_filled = df.fillna({"Age": df["Age"].mean(), "Salary": df["Salary"].median()})
print("\nAfter Handling Missing Data:\n", df_filled)
```

🔹 **Common Ways to Handle Missing Data**

- Drop missing values → df.dropna()
- Fill with a value (mean, median, mode, or custom) → df.fillna(value)
- Interpolate values → df.interpolate()
- Flag missing values → create a new column to indicate where data was missing

✅ In short:

Handling missing data in Pandas is crucial because it ensures accuracy, consistency, and usability of datasets, allowing you to perform reliable analysis and build robust models.

Q14. What are the benefits of using Plotly for data visualization?

Explanation:

Plotly is a popular Python library for interactive data visualization, and it offers several benefits over traditional static plotting libraries.

🔹 **Benefits of Using Plotly**

1.Interactive Visualizations

- Built-in zooming, panning, and hover tooltips.
- Lets users explore data instead of just looking at a static chart.

2.Wide Variety of Charts

- Supports simple plots (line, bar, scatter) and advanced ones (heatmaps, 3D plots, maps, financial charts, etc.).

3.Web & Dashboard Friendly

- Works seamlessly in Jupyter Notebooks and can be embedded into web apps using Dash (a Plotly framework).

4.High-Quality Graphics

- Produces modern, publication-quality charts with attractive default styles.

5.Easy to Use

- High-level syntax (plotly.express) makes it easy to create complex charts in just a few lines.

6.Customization

- Highly flexible with options to tweak layout, colors, annotations, and interactivity.

7.Cross-Platform Sharing

- Plots can be exported to HTML and shared without needing Python installed.

🔹 **Example**

```
import plotly.express as px

# Sample dataset
df = px.data.iris()

# Interactive scatter plot
fig = px.scatter(df, x="sepal_width", y="sepal_length", color="species",
                 size="petal_length", hover_data=["petal_width"])
fig.show()
```

✅ This plot is fully interactive: you can hover over points, zoom in, and filter by species.

🔹 **In Short**

Plotly is best when you want interactive, web-friendly, and visually appealing charts — ideal for dashboards, presentations, and data exploration.

Q15. How does NumPy handle multidimensional arrays?

Explanation:

NumPy handles multidimensional arrays using its core data structure called the ndarray (N-dimensional array). This allows you to work efficiently with 1D (vectors), 2D (matrices), 3D (cubes), or higher-dimensional (tensors) arrays.

🔹 How NumPy Handles Multidimensional Arrays

1.Shape

- Defined by a tuple that shows the size along each dimension.
- Example: (3, 4) means 3 rows and 4 columns.

2.Indexing & Slicing

- Works similar to Python lists but extended to multiple dimensions.
- Example: arr[1, 2] gives the element in row 1, column 2.

3.Vectorized Operations

- Operations (like addition, multiplication) apply element-wise across all dimensions.

4.roadcasting

- Allows operations between arrays of different shapes by "stretching" smaller arrays.

5.Axis-based operations

- Many functions (sum, mean, etc.) can be applied along a specific axis.
- Example: arr.sum(axis=0) ⟶ sum by columns, arr.sum(axis=1) ⟶ sum by rows.

🔹 **Example**

```
import numpy as np

# Create a 2D array (3x3 matrix)
arr = np.array([[1, 2, 3],
                [4, 5, 6],
                [7, 8, 9]])

print("Shape:", arr.shape)         # (3, 3)
```

```
print("Element at [1,2]:", arr[1, 2])   # 6
print("Row sums:", arr.sum(axis=1))     # [ 6 15 24]
print("Column sums:", arr.sum(axis=0))  # [12 15 18]
```

✅ In short:

NumPy handles multidimensional arrays through its ndarray, which supports efficient storage, slicing, broadcasting, and vectorized operations across any number of dimensions — making it ideal for scientific and mathematical computing.

Q16. What is the role of Bokeh in data visualization?

Explanation:

The Bokeh library in Python is designed for creating interactive and web-friendly visualizations. Unlike static libraries like Matplotlib, Bokeh focuses on building interactive plots, dashboards, and data applications that can run directly in a browser.

🔹 **Role of Bokeh in Data Visualization**

1.Interactive Visualizations

- Provides tools for zooming, panning, tooltips, and linked brushing (highlighting across multiple plots).

2.Web Integration

- Plots can be easily embedded in web applications (using HTML, JavaScript).
- Works well with frameworks like Flask or Django for dashboards.

3.Scalable Data Handling

- Supports visualization of large datasets using efficient rendering techniques (like WebGL).

4.Custom Dashboards

- Comes with Bokeh Server, which allows building real-time, interactive dashboards with widgets (sliders, dropdowns, etc.).

5.Flexible Output

- Visualizations can be exported to HTML, PNG, or notebooks, making them easy to share.

🔹 **Example**

```
from bokeh.plotting import figure, show

# Create a simple interactive line chart
p = figure(title="Bokeh Line Chart", x_axis_label='x', y_axis_label='y')

x = [1, 2, 3, 4, 5]
y = [6, 7, 2, 4, 5]

p.line(x, y, line_width=2)

show(p)  # Opens interactive plot in browser
```

✅ This will open an interactive line chart in your browser, where you can zoom and hover over points.

🔹 In Short:

Bokeh's role in data visualization is to make it easy to create interactive, web-ready, and scalable plots—perfect for dashboards, big data exploration, and real-time applications.

Q17. Explain the difference between apply() and map() in Pandas.

Explanation:

In Pandas, both apply() and map() are used to apply functions, but they work at different levels and have different flexibility.

🔹 **Difference Between apply() and map()**

1. map()

- Works element-wise on a Series (1D).
- Can take a function, dictionary, or a Series for mapping.

- Simple and efficient, but limited to 1D only.

✅ Example:

```
import pandas as pd

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

# Using map with a function
print(s.map(lambda x: x ** 2))

# Using map with a dictionary
print(s.map({1: 'A', 2: 'B', 3: 'C'}))


Output:

0     1
1     4
2     9
3    16
dtype: int64

0      A
1      B
2      C
3    NaN
dtype: object
```

2. apply()

- Works on both Series (1D) and DataFrames (2D).
- Can apply a function along rows (axis=1) or columns (axis=0).

- More flexible and powerful than map().

✅ Example:

```
df = pd.DataFrame({
    'A': [1, 2, 3],
    'B': [4, 5, 6]
})

# Apply on Series (column)
print(df['A'].apply(lambda x: x ** 2))

# Apply on DataFrame (row-wise)
print(df.apply(lambda row: row['A'] + row['B'], axis=1))


Output:

0    1
1    4
2    9
Name: A, dtype: int64

0    5
1    7
2    9
dtype: int64
```

🔹 Summary

Feature	`map()`	`apply()`
Works on	**Series only**	Series & DataFrame
Level of operation	Element-wise	Element-wise OR row/column-wise

Feature	`map()`	`apply()`
Input types supported	Function, dict, Series	Function only
Flexibility	Limited	Very flexible

👉 Rule of Thumb:

- Use map() when working with a single Series.
- Use apply() when working with DataFrames or when you need more flexibility.

Q18. What are some advanced features of NumPy?

Explanation:

NumPy isn't just about arrays and basic math — it has several advanced features that make it powerful for scientific computing, machine learning, and data analysis. Here are some of the most important ones:

🔹 **Advanced Features of NumPy**

1.Broadcasting

- Allows operations on arrays of different shapes without making explicit copies.
- Example: adding a scalar or a smaller array to a larger one.

```
import numpy as np
A = np.array([[1, 2, 3], [4, 5, 6]])
b = np.array([10, 20, 30])
print(A + b)   # b is broadcast across rows
```

2.Vectorized Operations

- Apply operations element-wise without explicit loops.
- Much faster due to underlying C implementation.

```
arr = np.array([1, 2, 3, 4])
print(arr ** 2)  # Vectorized squaring
```

3.Masked Arrays

- Handle missing or invalid data by “masking” values.
- Useful for scientific datasets with gaps.

```
import numpy.ma as ma
arr = np.array([1, 2, 3, -999, 5])
masked = ma.masked_equal(arr, -999)
print(masked.mean())  # Ignores the -999
```

4.Memory Mapping (memmap)

- Work with very large arrays stored on disk without loading them entirely into memory.

```
mmap_arr = np.memmap('data.dat', dtype='float32', mode='w+', shape=(10000, 10000))
mmap_arr[0, 0] = 42
```

5.Linear Algebra Routines

- Built-in support for matrix operations: determinants, eigenvalues, singular value decomposition (SVD), etc.

```
A = np.array([[1, 2], [3, 4]])
print(np.linalg.det(A))   # Determinant
print(np.linalg.eig(A))   # Eigenvalues & eigenvectors
```

6.FFT (Fast Fourier Transform)

- Perform frequency analysis using np.fft.

```
signal = np.array([0, 1, 0, -1])
print(np.fft.fft(signal))
```

7.Random Module (numpy.random)

- Advanced random sampling, distributions, shuffling, and simulation.

```
rng = np.random.default_rng()
print(rng.normal(loc=0, scale=1, size=5))  # Normal distribution
```

8.Structured & Record Arrays

- Store heterogeneous data types (like a table with named columns).

```
data = np.array([(1, 'Alice', 25.5), (2, 'Bob', 30.0)],
                dtype=[('id', 'i4'), ('name', 'U10'), ('age', 'f4')])
print(data['name'])
```

9.Advanced Indexing & Boolean Masking

- Extract elements conditionally or with complex indices.

```
arr = np.arange(10)
print(arr[arr % 2 == 0])  # Select even numbers
```

10.Integration with Other Libraries

- NumPy arrays are the foundation of Pandas, SciPy, Scikit-learn, TensorFlow, and PyTorch.
- This makes it the backbone of the Python scientific stack.

✅ In short:

NumPy's advanced features like broadcasting, vectorization, memory mapping, linear algebra, FFTs, and masked arrays make it a high-performance tool for scientific and analytical computing.

Q19. How does Pandas simplify time series analysis?

Explanation:

Pandas is one of the best tools for time series analysis in Python, because it provides built-in functions and data structures that make working with dates, times, and frequency-based data very convenient.

🔹 **How Pandas Simplifies Time Series Analysis**

1.Date and Time Handling (datetime64)

- Pandas automatically converts date strings into datetime objects.
- Easy to extract year, month, day, weekday, etc.

```
import pandas as pd

dates = pd.to_datetime(["2023-01-01", "2023-02-01", "2023-03-01"])
print(dates.month)   # [1, 2, 3]
```

2.Date Range Generation

- Create sequences of dates with different frequencies (D, M, Y, H, etc.).

```
pd.date_range(start="2023-01-01", periods=5, freq="D")
# Generates daily dates
```

3.Indexing with Dates (DatetimeIndex)

- Time series data often uses dates as the index.
- Allows slicing and filtering by year, month, or exact date.

```
ts = pd.Series([10, 20, 30], index=pd.date_range("2023-01-01", periods=3))
print(ts["2023-01"])   # Select all January data
```

4.Resampling & Frequency Conversion

- Convert data between different time frequencies (e.g., daily → monthly).
- Supports aggregation like mean, sum, etc.

```
ts = pd.Series([1, 2, 3, 4, 5], index=pd.date_range("2023-01-01", periods=5, freq="D"))
print(ts.resample("2D").mean())  # Resample every 2 days
```

5.Rolling & Moving Window Calculations

- Calculate rolling averages, sums, or other statistics over a time window.

```
ts = pd.Series([1, 2, 3, 4, 5], index=pd.date_range("2023-01-01", periods=5))
print(ts.rolling(window=3).mean())
```

6.Shifting & Lagging Data

- Shift data forward or backward in time (useful for lag features).

```
print(ts.shift(1))  # Shift by 1 day
```

7.Time Zone Support

- Convert timestamps between different time zones easily.

```
tz_series = ts.tz_localize("UTC").tz_convert("US/Eastern")
print(tz_series)
```

8.Integration with Other Libraries

- Works seamlessly with Matplotlib and Seaborn for time series plots.
- Often used with statsmodels for forecasting.

✅ In short:

Pandas simplifies time series analysis with its DatetimeIndex, resampling, rolling windows, shifting, and timezone handling, making it very powerful for tasks like financial analysis, forecasting, and trend detection.

Q20. What is the role of a pivot table in Pandas?

Explanation:

A pivot table in Pandas is used to summarize, aggregate, and reorganize data in a tabular form, similar to Excel pivot tables. It allows you to reshape data, calculate summary statistics, and analyze large datasets efficiently.

🔹 **Role of Pivot Tables in Pandas**

1.Data Aggregation

- Compute sums, averages, counts, or other statistics for specific groups.

2.Data Reshaping

- Transform long-form data into a matrix/table format for easier analysis.

3.Multidimensional Analysis

- Supports multiple rows and columns (multi-indexing) to view data from different angles.

4.Quick Insights

- Helps in exploratory data analysis (EDA) by summarizing large datasets in a readable format.

🔹 **Example**

```
import pandas as pd

data = {
    "Department": ["HR", "HR", "IT", "IT", "Finance", "Finance"],
    "Employee": ["A", "B", "C", "D", "E", "F"],
    "Salary": [40000, 42000, 50000, 55000, 60000, 58000]
}
```

```
df = pd.DataFrame(data)

# Create pivot table to calculate average salary per department
pivot = pd.pivot_table(df, values="Salary", index="Department", aggfunc="mean")
print(pivot)
```

Output:

```
            Salary
Department
Finance    59000.0
HR         41000.0
IT         52500.0
```

🔹 **Additional Features**

- Can aggregate multiple values at once: aggfunc=["mean", "sum", "max"]
- Can use multiple indexes or columns for more complex summaries.
- Can fill missing values using fill_value parameter.

✅ In short:

A pivot table in Pandas is a powerful tool for summarizing and reshaping data, allowing you to quickly calculate statistics and gain insights from large datasets.

Q21. Why is NumPy's array slicing faster than Python's list slicing?

Explanation:

NumPy's array slicing is faster than Python's list slicing because of how data is stored and accessed internally. Here's why:

🔹 **Reasons for NumPy's Faster Slicing**

1.Contiguous Memory Storage

- NumPy arrays are stored in contiguous blocks of memory (like C arrays), whereas Python lists store pointers to separate Python objects.
- This allows NumPy to access a block of memory directly, reducing overhead.

2.Vectorized Operations

- NumPy slices don't copy data by default—they create views on the original array.
- Python lists always create a new list when slicing, which requires allocating new memory and copying elements.

3.Lower-Level Implementation

- NumPy is implemented in C, which is much faster than Python's interpreted loops and object handling.

4.Homogeneous Data Types

- All elements in a NumPy array are of the same type, so operations can be optimized.
- Python lists can contain mixed types, so Python must check types at runtime.

🔹 **Example**

```
import numpy as np
import time

# NumPy array
arr = np.arange(1000000)
start = time.time()
slice_arr = arr[100:10000]  # creates a view, no copying
end = time.time()
print("NumPy slice time:", end - start)

# Python list
lst = list(range(1000000))
start = time.time()
slice_lst = lst[100:10000]  # creates a new list, copying data
```

```
end = time.time()
print("Python list slice time:", end - start)
```

👉 Output: NumPy slicing is significantly faster because it doesn't copy data and accesses memory directly.

✅ In short:

NumPy slicing is faster than Python list slicing because arrays are contiguous, homogeneous, and use low-level C operations, often creating views instead of copies, whereas Python lists involve pointer dereferencing and element copying.

Q22. What are some common use cases for Seaborn?

Explanation:

Seaborn is a high-level Python visualization library built on Matplotlib, designed for statistical and exploratory data analysis. Here are some common use cases:

🔹 1. Visualizing Distributions of Data

- Understand the spread and shape of numeric data.
- Useful for spotting skewness, outliers, or multi-modality.

Functions: histplot(), kdeplot(), displot(), boxplot(), violinplot()

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

tips = sns.load_dataset("tips")
sns.histplot(tips["total_bill"], kde=True)
plt.show()
```

🔹 2. Exploring Relationships Between Variables

- Investigate how two variables are correlated or related.
- Useful for regression, trend analysis, or spotting patterns.

Functions: scatterplot(), regplot(), lmplot(), pairplot()

```
sns.scatterplot(x="total_bill", y="tip", data=tips, hue="time")
plt.show()
```

🔹 3. Correlation and Heatmaps

- isualize correlation matrices or values in a grid format.
- Useful for identifying strong or weak relationships between variables.

Function: heatmap()

```
corr = tips.corr()
sns.heatmap(corr, annot=True, cmap="coolwarm")
plt.show()
```

🔹 4. Categorical Data Analysis

- Compare distributions or relationships across different categories.

Functions: countplot(), barplot(), boxplot(), violinplot()

```
sns.countplot(x="day", data=tips, hue="sex")
plt.show()
```

🔹 5. Pairwise Plots for Multivariate Analysis

- Quickly explore relationships among multiple numeric variables.

Function: pairplot()

- Often used with the hue parameter to differentiate categories.

🔹 6. Time Series and Trend Analysis

- Plot trends over time with confidence intervals.

Functions: lineplot(), relplot()

```
sns.lineplot(x="size", y="total_bill", data=tips)
plt.show()
```

- ✅ In Short
  - Seaborn is commonly used for:
  - Distribution analysis
  - Correlation and relationships
  - Categorical comparisons
  - Multivariate exploration
  - Trend visualization

It's especially useful for quick, attractive, and informative statistical plots in exploratory data analysis.

Practical Question

Q1. How do you create a 2D NumPy array and calculate the sum of each row?

Explanation:

You can create a 2D NumPy array using np.array() or np.zeros()/np.ones()/np.random functions, and then calculate the sum of each row using the sum() function with the axis parameter.

- **Example**

```
import numpy as np

# Create a 2D NumPy array (3 rows, 4 columns)
```

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

# Calculate the sum of each row
row_sums = arr.sum(axis=1)  # axis=1 sums along columns for each row
print("2D Array:\n", arr)
print("Sum of each row:", row_sums)
```

Output:

```
2D Array:
 [[ 1  2  3  4]
  [ 5  6  7  8]
  [ 9 10 11 12]]
Sum of each row: [10 26 42]
```

🔹 **Explanation**

- axis=0 ⟶ operate column-wise (sum of each column)
- axis=1 ⟶ operate row-wise (sum of each row)
- NumPy efficiently performs these operations using vectorized computations.

Q2. Write a Pandas script to find the mean of a specific column in a DataFrame.

Explanation:

```
import pandas as pd

# Sample data
data = {
    "Name": ["Alice", "Bob", "Charlie", "David"],
```

```
    "Age": [25, 30, 22, 28],
    "Salary": [50000, 60000, 45000, 55000]
}

# Create DataFrame
df = pd.DataFrame(data)

# Find mean of the 'Age' column
mean_age = df["Age"].mean()
print("Mean Age:", mean_age)

# Find mean of the 'Salary' column
mean_salary = df["Salary"].mean()
print("Mean Salary:", mean_salary)
```

Output:

```
Mean Age: 26.25
Mean Salary: 52500.0
```

🔹 **Explanation**

- df["ColumnName"] → selects the column.
- .mean() → calculates the mean (average) of that column.

You can also calculate the mean for all numeric columns at once using:

```
df.mean()
```

This will return a Series with the mean of each numeric column.

Q3. Create a scatter plot using Matplotlib.

Explanation:

Here's a simple example of creating a scatter plot using Matplotlib in Python:

```
import matplotlib.pyplot as plt

# Sample data
x = [1, 2, 3, 4, 5]
y = [10, 15, 13, 17, 20]

# Create scatter plot
plt.scatter(x, y, color='blue', marker='o', s=100)  # s = size of points
plt.title("Sample Scatter Plot")
plt.xlabel("X-axis")
plt.ylabel("Y-axis")
plt.grid(True)

# Display the plot
plt.show()
```

🔹 **Explanation**

- plt.scatter(x, y) ⟶ creates a scatter plot with x and y coordinates.
- color ⟶ color of the points.
- marker ⟶ shape of the points ('o', 's', '^', etc.).
- s ⟶ size of the points.
- plt.title(), plt.xlabel(), plt.ylabel() ⟶ add labels and title.
- plt.grid(True) ⟶ adds a grid for better readability.

Q4. How do you calculate the correlation matrix using Seaborn and visualize it with a heatmap?

Explanation:

You can calculate the correlation matrix in Pandas and then visualize it using Seaborn's heatmap. Here's a step-by-step example:

🔹 Example

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

# Sample dataset
data = {
    "Math": [85, 90, 78, 92, 88],
    "Science": [80, 85, 75, 95, 89],
    "English": [78, 82, 88, 90, 85],
    "History": [70, 75, 72, 80, 78]
}

df = pd.DataFrame(data)

# Calculate correlation matrix
corr_matrix = df.corr()
print("Correlation Matrix:\n", corr_matrix)

# Visualize using Seaborn heatmap
sns.heatmap(corr_matrix, annot=True, cmap="coolwarm", linewidths=0.5)
plt.title("Correlation Heatmap")
plt.show()
```

🔹 Explanation

1.df.corr() ⟶ computes the Pearson correlation between numeric columns.

2.sns.heatmap() ⟶ visualizes the correlation matrix:

- annot=True ⟶ shows correlation values on the heatmap.
- cmap="coolwarm" ⟶ color gradient for better readability.
- linewidths=0.5 ⟶ adds lines between cells.

✅ **In short:**

- Use Pandas .corr() to calculate the correlation matrix.
- Use Seaborn's heatmap() to visualize relationships, making it easy to spot strong positive, negative, or no correlation.

Q5. Generate a bar plot using Plotly.

Explanation:

Here's a simple example of creating a bar plot using Plotly in Python:

🔹 **Example Using Plotly Express**

```
import plotly.express as px

# Sample data
data = {
    "Fruits": ["Apples", "Bananas", "Cherries", "Dates"],
    "Quantity": [10, 15, 7, 12]
}

# Create bar plot
fig = px.bar(data, x="Fruits", y="Quantity",
             title="Fruit Quantities",
             color="Quantity",         # optional: color by value
             text="Quantity")          # display value on bars
```

```
# Show interactive plot
fig.show()
```

🔹 **Explanation**

- px.bar() ⟶ creates a bar chart.
- x ⟶ categories (horizontal axis).
- y ⟶ numeric values (height of bars).
- color ⟶ optional, for coloring bars based on values.
- text ⟶ optional, shows value labels on bars.
- fig.show() ⟶ renders an interactive plot in Jupyter Notebook or browser.

✅ In short:

Plotly makes it easy to create interactive bar charts with hover info, zooming, and customizable colors in just a few lines.

Q6. Create a DataFrame and add a new column based on an existing column.

Explanation:

Here's a simple example showing how to create a Pandas DataFrame and add a new column derived from an existing column:

🔹 **Example**

```
import pandas as pd

# Create a sample DataFrame
data = {
    "Name": ["Alice", "Bob", "Charlie", "David"],
    "Salary": [50000, 60000, 45000, 55000]
}

df = pd.DataFrame(data)
```

```
print("Original DataFrame:\n", df)

# Add a new column 'Tax' as 10% of 'Salary'
df["Tax"] = df["Salary"] * 0.10
print("\nDataFrame after adding 'Tax' column:\n", df)
```

Output:

```
Original DataFrame:
      Name  Salary
0    Alice   50000
1      Bob   60000
2  Charlie   45000
3    David   55000

DataFrame after adding 'Tax' column:
      Name  Salary     Tax
0    Alice   50000  5000.0
1      Bob   60000  6000.0
2  Charlie   45000  4500.0
3    David   55000  5500.0
```

- **Explanation**
  - df["NewColumn"] = ... ⟶ adds a new column to the DataFrame.
  - You can perform arithmetic, conditional operations, or functions on existing columns to create new ones.

Q7. Write a program to perform element-wise multiplication of two NumPy arrays.

Explanation:

Here's a simple Python program to perform element-wise multiplication of two NumPy arrays:

- **Example**

```
import numpy as np

# Create two NumPy arrays
arr1 = np.array([1, 2, 3, 4])
arr2 = np.array([5, 6, 7, 8])

# Element-wise multiplication
result = arr1 * arr2  # or np.multiply(arr1, arr2)

print("Array 1:", arr1)
print("Array 2:", arr2)
print("Element-wise multiplication:", result)
```

Output:

```
Array 1: [1 2 3 4]
Array 2: [5 6 7 8]
Element-wise multiplication: [ 5 12 21 32]
```

🔹 **Explanation**

- arr1 * arr2 performs element-wise multiplication directly.
- np.multiply(arr1, arr2) does the same and is useful when you want a function call instead of operator.
- Both arrays must have the same shape, or they must be broadcastable.

Q8. Create a line plot with multiple lines using Matplotlib.

Explanation:

Here's a simple example of creating a line plot with multiple lines using Matplotlib:

🔹 **Example**

```
import matplotlib.pyplot as plt

# Sample data
x = [1, 2, 3, 4, 5]
y1 = [10, 15, 13, 17, 20]
y2 = [8, 12, 15, 19, 22]

# Create line plot with multiple lines
plt.plot(x, y1, marker='o', label='Line 1', color='blue')
plt.plot(x, y2, marker='s', label='Line 2', color='red')

# Add title and labels
plt.title("Multiple Line Plot")
plt.xlabel("X-axis")
plt.ylabel("Y-axis")
plt.legend()  # Show legend
plt.grid(True)

# Display the plot
plt.show()
```

🔹 **Explanation**

- plt.plot(x, y, ...) ⟶ plots a line for the given data.
- marker ⟶ shape of the points ('o', 's', '^', etc.).
- label ⟶ used in the legend to identify each line.
- plt.legend() ⟶ displays the legend.
- Multiple calls to plt.plot() add multiple lines to the same axes.

✅ In short:

Matplotlib allows plotting multiple lines on the same plot easily by calling plt.plot() for each dataset and using labels to differentiate them.

Q9. Generate a Pandas DataFrame and filter rows where a column value is greater than a threshold.

Explanation:

Here's a simple example of creating a Pandas DataFrame and filtering rows based on a column value exceeding a threshold:

🔹 **Example**

```
import pandas as pd

# Create a sample DataFrame
data = {
    "Name": ["Alice", "Bob", "Charlie", "David"],
    "Age": [25, 30, 22, 28],
    "Salary": [50000, 60000, 45000, 55000]
}

df = pd.DataFrame(data)
print("Original DataFrame:\n", df)

# Filter rows where Salary > 50000
filtered_df = df[df["Salary"] > 50000]
print("\nFiltered DataFrame (Salary > 50000):\n", filtered_df)
```

Output:

```
Original DataFrame:
      Name  Age  Salary
0    Alice   25   50000
1      Bob   30   60000
2  Charlie   22   45000
```

```
3    David   28   55000

Filtered DataFrame (Salary > 50000):
    Name  Age  Salary
1    Bob   30   60000
3  David   28   55000
```

🔹 **Explanation**

- df["Salary"] > 50000 ⟶ creates a Boolean mask for the condition.
- df[condition] ⟶ selects only the rows where the condition is True.
- You can apply any condition (<, >=, !=, etc.) on numeric or string columns.

Q10. Create a histogram using Seaborn to visualize a distribution.

Explanation:

Here's a simple example of creating a histogram using Seaborn to visualize the distribution of data:

🔹 **Example**

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

# Sample data
ages = [22, 25, 29, 21, 24, 30, 27, 26, 28, 23, 25, 24, 26]

# Create histogram
sns.histplot(ages, bins=5, kde=True, color='skyblue')

# Add title and labels
plt.title("Age Distribution")
plt.xlabel("Age")
```

```
plt.ylabel("Frequency")

# Show plot
plt.show()
```

🔹 **Explanation**

- sns.histplot(data) ⟶ creates a histogram.
- bins ⟶ number of bins to divide the data into.
- kde=True ⟶ overlays a Kernel Density Estimate (smooth curve showing distribution).
- color ⟶ sets the color of the bars.
- Matplotlib functions (plt.title(), plt.xlabel(), plt.ylabel()) are used to label the plot.

✅ In short:

Seaborn's histplot() makes it easy to visualize the distribution of numeric data, and adding kde=True gives a smooth estimate of the distribution.

Q11. Perform matrix multiplication using NumPy.

Explanation:

Here's how you can perform matrix multiplication using NumPy. Unlike element-wise multiplication, matrix multiplication follows linear algebra rules.

🔹 **Example**

```
import numpy as np

# Define two 2D arrays (matrices)
A = np.array([[1, 2],
              [3, 4]])
```

```
B = np.array([[5, 6],
              [7, 8]])

# Method 1: Using np.dot()
result1 = np.dot(A, B)

# Method 2: Using the @ operator (Python 3.5+)
result2 = A @ B

print("Matrix A:\n", A)
print("Matrix B:\n", B)
print("Matrix multiplication result (np.dot):\n", result1)
print("Matrix multiplication result (@ operator):\n", result2)
```

Output:

```
Matrix A:
 [[1 2]
  [3 4]]
Matrix B:
 [[5 6]
  [7 8]]
Matrix multiplication result (np.dot):
 [[19 22]
  [43 50]]
Matrix multiplication result (@ operator):
 [[19 22]
  [43 50]]
```

🔹 **Explanation**

- np.dot(A, B) ⟶ performs standard matrix multiplication.

- A @ B ⟶ shorthand operator for matrix multiplication.
- Shape rule: If A is (m, n), then B must be (n, p); result is (m, p).

✅ In short:

NumPy provides np.dot() and the @ operator for matrix multiplication, which is essential for linear algebra, machine learning, and scientific computing.

Q12. Use Pandas to load a CSV file and display its first 5 rows.

Explanation:

Here's a simple example of how to load a CSV file using Pandas and display the first 5 rows:

🔹 **Example**

```
import pandas as pd

# Load CSV file
df = pd.read_csv("data.csv")  # replace 'data.csv' with your file path

# Display first 5 rows
print(df.head())
```

🔹 **Explanation**

- pd.read_csv("file.csv") ⟶ reads a CSV file into a DataFrame.
- df.head() ⟶ shows the first 5 rows by default.
- You can pass a number to head(), e.g., df.head(10) to display the first 10 rows.

✅ In short:

Pandas makes it easy to load and inspect CSV files. Using head() gives a quick overview of the dataset.

Q13. Create a 3D scatter plot using Plotly.

Explanation:

Here's an example of creating an interactive 3D scatter plot using Plotly:

- **Example Using Plotly Express**

```
import plotly.express as px
import pandas as pd

# Sample data
data = {
    "X": [1, 2, 3, 4, 5],
    "Y": [10, 15, 13, 17, 20],
    "Z": [5, 7, 6, 8, 9],
    "Category": ["A", "B", "A", "B", "A"]
}

df = pd.DataFrame(data)

# Create 3D scatter plot
fig = px.scatter_3d(
    df,
   x="X", y="Y", z="Z",
    color="Category",           # Color points by category
    size="Z",                   # Size points based on Z
    symbol="Category",          # Different marker symbols by category
    title="3D Scatter Plot Example"
)

# Show interactive plot
fig.show()
```

🔹 **Explanation**

- px.scatter_3d() ⟶ creates a 3D scatter plot.
- x, y, z ⟶ coordinates of points in 3D space.