hbhpgqjo6

September 18, 2025

1 DATA TOOLKIT

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

NumPy (short for Numerical Python) is a **powerful open-source library** in Python used for numerical computing. It provides tools for working with large, multi-dimensional arrays and matrices, along with a wide **collection of high-level mathematical functions** to operate on these arrays efficiently.

- A **Python library** that adds support for n-dimensional arrays (called ndarray).
- Provides **vectorized operations**, meaning operations are applied on whole arrays at once (instead of element by element).
- Written in C and optimized for speed, making it much faster than using plain Python lists for **numerical tasks**.

Why NumPy is widely used: 1.Efficient array operations

- NumPy arrays use less memory and are faster than Python lists.
- Example: Multiplying two arrays with * in NumPy is element-wise and optimized.

2.Mathematical functions

- Provides functions for linear algebra, statistics, Fourier transforms, random number generation, etc.

3.Integration with other libraries

- Core dependency for **pandas, scikit-learn, TensorFlow, PyTorch, Matplotlib, and many more**.
- Acts as the foundation of the **scientific Python ecosystem**.

4.Convenient slicing and indexing

- More powerful than **Python lists** (supports multi-dimensional slicing, boolean indexing, fancy indexing).

5.Broadcasting

- Allows **operations between arrays of different shapes** without explicitly writing loops.

6.Cross-platform

- Works across different **operating systems and hardware** (including GPU acceleration through libraries like CuPy).

Q2.How does broadcasting work in NumPy?

Broadcasting is a set of rules that NumPy follows when performing **arithmetic operations on arrays** with different shapes.

Instead of requiring arrays to be the exact same shape, NumPy tries to "stretch" the **smaller array across the larger one so that element-wise*** operations are possible without actually copying data.

Rules of Broadcasting - Compare their shapes from right to left.

- Dimensions are compatible if:

They are equal, or

One of them is 1.

- If one array has fewer dimensions, NumPy adds leading 1s to make the shapes match.
- If dimensions are still incompatible, NumPy raises an error.

Q3.What is a Pandas DataFrame?

A **Pandas DataFrame** is a **two-dimensional, tabular data structure** in the **pandas library** (built on top of NumPy).

It is like a **spreadsheet or SQL table in Python** — with rows and columns, where:

- **ROWS** → represent observations/records
- **COLUMNS** → represent features/attributes

Each column can hold different data types (integer, float, string, datetime, etc.)

KEY FEATURE OF DATAFRAME - 1.Labeled axes → Rows (index) and Columns (column names).

- **2.Heterogeneous data** → Different data types in different columns.
- **3.Size mutable** → Can add or drop rows/columns.
- **4.Data alignment** → Handles missing data gracefully (NaN).
- **5.Built-in methods** → For filtering, grouping, aggregation, merging, reshaping, etc.

EXAMPLE:

[ ]:
```python
import pandas as pd

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

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

print(df)
```

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

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

The groupby() method in Pandas is used to **split data into groups** based on the values in one or more columns, and then apply **aggregation, transformation, or filtering operations** on those groups.

It follows the "**split → apply → combine**" process:

- **1.Split** – Divide the data into groups (by column values).
- **2.Apply** – Apply a function (like sum, mean, count, etc.) to each group.
- **3.Combine** – Merge the results back into a DataFrame.

SYNTAX:

df.groupby('column_name')

df.groupby(['col1', 'col2'])

EXAMPLE:

[ ]:
```python
import pandas as pd

data = {
    "Department": ["HR", "HR", "IT", "IT", "Finance"],
    "Employee": ["Alice", "Bob", "Charlie", "David", "Eva"],
    "Salary": [50000, 55000, 60000, 65000, 70000]
}

df = pd.DataFrame(data)

# Group by Department and calculate average salary
result = df.groupby("Department")["Salary"].mean()

print(result)
```

```
Department
Finance    70000.0
HR         52500.0
IT         62500.0
Name: Salary, dtype: float64
```

Q5.Why is Seaborn preferred for statistical visualizations?

Seaborn is a Python data visualization library built on top of Matplotlib. It is preferred for **statistical visualizations because it provides a high-level, easy-to-use interface and comes with built-in support** for statistical plots.

- **KEY REASON SEABORN IS PREPARERD:**

1.Simpler Syntax & High-Level API

- Seaborn lets you create complex statistical plots with just one line of code, whereas **Matplotlib** often requires many lines.

2.Beautiful Default Styles

- **Seaborn** has attractive, **modern default themes that make plots** look professional without extra formatting.

3.Built-in Statistical Functions

- Automatically handles statistical **estimation and visualization (e.g., confidence intervals**, regression lines).
- **Example**: sns.regplot() adds regression line + confidence interval automatically.

4.ntegration with Pandas DataFrames

- Works directly with Pandas DataFrames and column names, reducing the need for manual indexing.

5.Specialized Statistical Plots

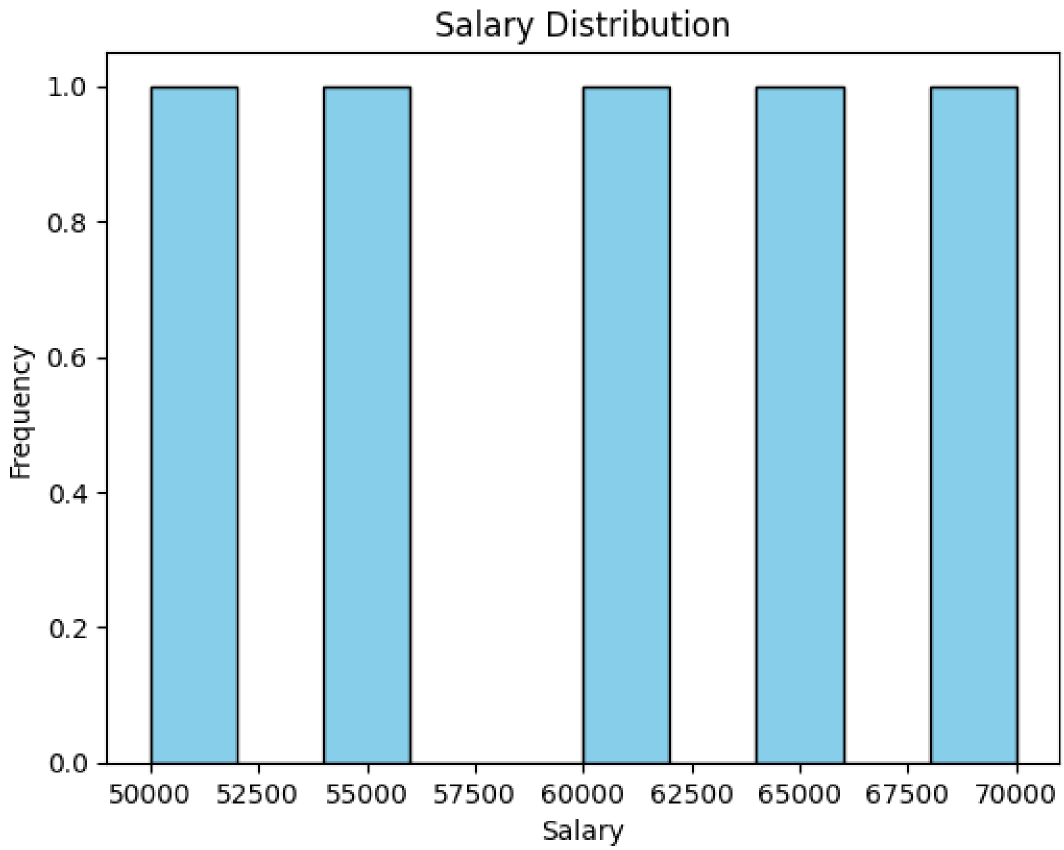
- Provides advanced plots that Matplotlib doesn't have out-of-the-box, like:
- Heatmaps (sns.heatmap)
- Pair plots (sns.pairplot)
- Violin plots (sns.violinplot)
- Distribution plots (sns.histplot, sns.kdeplot)

6.Automatic Handling of Categorical Data

- Seaborn makes it easy to **compare categories visually** (bar plots, box plots, swarm plots, etc.).

EXAMPLE:

[ ]:
```
import matplotlib.pyplot as plt
plt.hist(df["Salary"], bins=10, color="skyblue", edgecolor="black")
plt.xlabel("Salary")
plt.ylabel("Frequency")
plt.title("Salary Distribution")
plt.show()
```



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

Differences Between NumPy Arrays and Python Lists:

Feature	**Python List**	**NumPy Array (`ndarray`)**
Data Type	Can store **heterogeneous data** (int, float, string, etc.) in the same list	Stores **homogeneous data** (all elements must be of the same type)
Memory Usage	Stores data as **objects**, so it is less memory-efficient	Stores data in **contiguous blocks** of memory → more efficient
Performance	Slower for numerical operations (uses Python loops)	Much faster (vectorized operations implemented in C)
Functionality	Only basic operations (append, pop, slicing)	Supports advanced mathematical, linear algebra, statistical operations
Dimension Support	1D only (list of lists for 2D, but clunky)	Supports **n-dimensional arrays** (matrix, tensor, etc.)
Broadcasting	Not supported	Fully supports **broadcasting** (operations on arrays of different shapes)
Element-wise Operations	Must use loops or list comprehensions	Directly supports element-wise operations (e.g., `a + b`)

Feature	**Python List**	**NumPy Array (`ndarray`)**
Integration	General-purpose	Backbone for data science libraries (Pandas, Scikit-learn, TensorFlow, etc.)

EXAMPLE - PYTHON LIST

[ ]:
```
lst = [1, 2, 3, 4]
result = [x*2 for x in lst]
print(result)
```

```
[2, 4, 6, 8]
```

- **NUMPY ARRAY**

[ ]:
```
import numpy as np

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

```
[2 4 6 8]
```

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

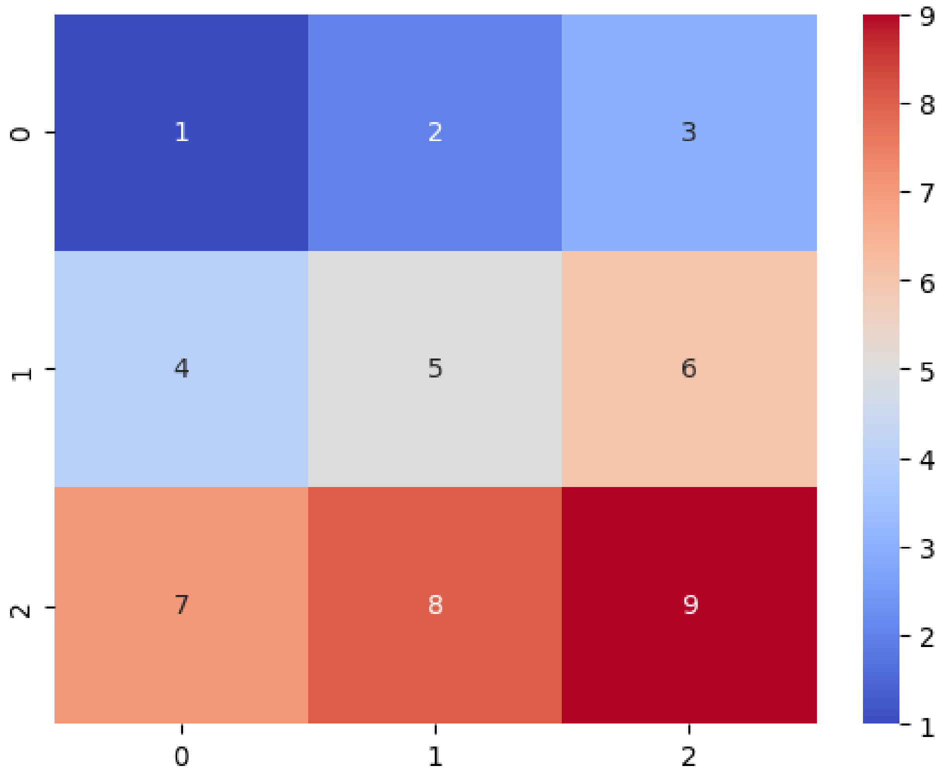
A **heatmap is a data visualization technique** that uses color shading to represent values in a 2D matrix or table.

- Each cell in the table is *colored** based on its value.
- **Darker or brighter colors** usually represent higher or lower values (depending on the color scale).

In Python, heatmaps are commonly created with **Seaborn (sns.heatmap) or Matplotlib.**

EXAMPLE

[ ]:
```
import seaborn as sns
import numpy as np
import matplotlib.pyplot as plt

# Sample data
data = np.array([[1, 2, 3],
                 [4, 5, 6],
                 [7, 8, 9]])

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



When Should a Heatmap Be Used:

1.Correlation Analysis

- To show relationships between variables in a dataset.

sns.heatmap(df.corr(), annot=True, cmap="viridis")

2.Visualizing Matrices

- Great for showing confusion matrices in machine learning.

3.Highlighting Patterns

- Easy to spot trends, clusters, or anomalies in data (e.g., sales over time, temperature variations).

4.Comparisons in Large Data

- Makes large numeric datasets easier to interpret visually.

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

A vectorized operation means **performing an operation** on an entire array (or batch of data) at once, without writing explicit **Python loops**.

- NumPy implements these operations in **optimized C code** under the hood.

- This makes them much faster and more concise than looping through elements in **pure Python.**

EXAMPLE

Without Vectorization (using a loop)

[ ]:
```
numbers = [1, 2, 3, 4]
result = []
for n in numbers:
    result.append(n * 2)

print(result)  # [2, 4, 6, 8]
```

```
[2, 4, 6, 8]
```

With Vectorization (NumPy)

[ ]:
```
import numpy as np

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

print(result)  # [2 4 6 8]
```

```
[2 4 6 8]
```

Why Vectorized Operations are Important - 1.Speed → Runs in C (much faster than Python loops).

- **2.Simplicity** → Cleaner, more readable code.
- **3.Memory Efficiency** → No need for intermediate lists.
- **4.Mathematical Expressiveness** → Code looks like real math equations

Q9.How does Matplotlib differ from Plotly?

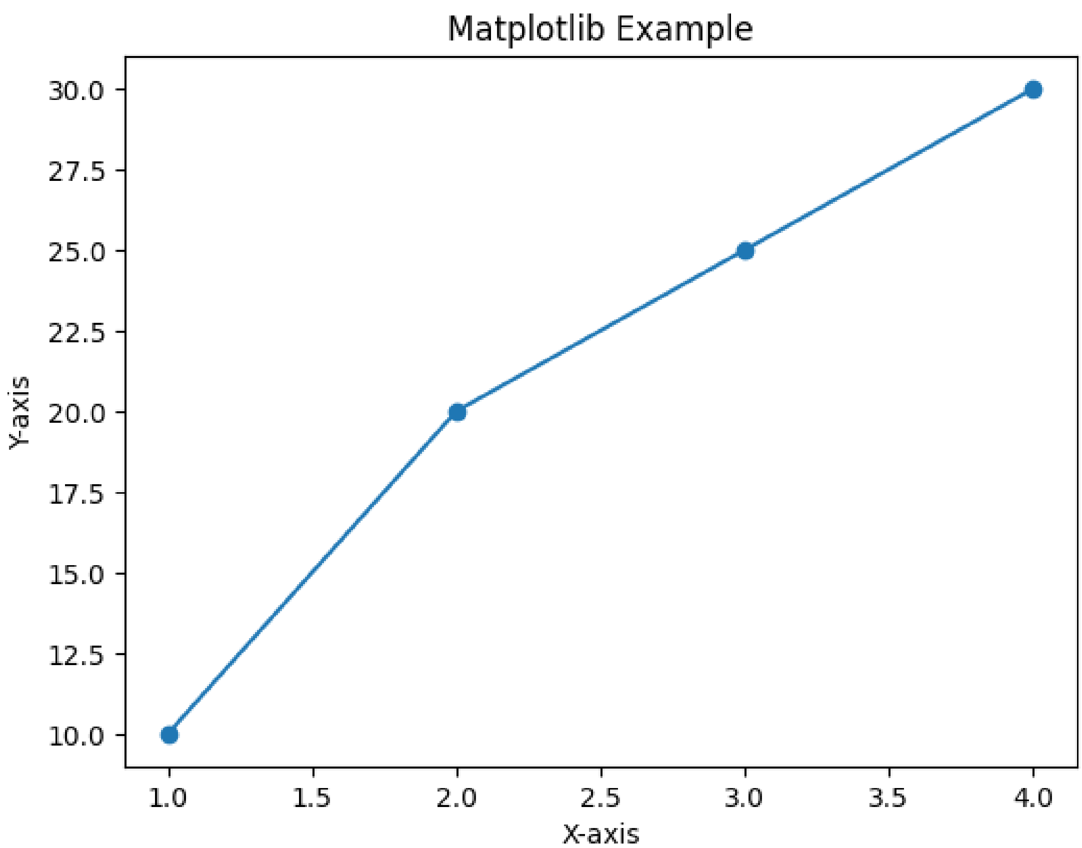
- **KEY DIFFERENCES BETWEEN MATPLOTLIB & PLOTLY**

Feature	**Matplotlib**	**Plotly**
Type	Low-level, static plotting library	High-level, interactive plotting library
Interactivity	Mostly static (can use `mpl_interactions` or `%matplotlib notebook`, but limited)	Fully interactive (zoom, pan, hover tooltips, clickable legends)
Ease of Use	Requires more code for styling and customization	Easier for interactive dashboards and quick interactive plots
Customization	Very flexible, but verbose	Limited compared to Matplotlib, but sufficient for most
Integration	Works well with Pandas, NumPy, Seaborn	Works with Pandas, NumPy, Dash (for web apps)
Output	Best for static plots (PDFs, PNGs, scientific papers)	Best for interactive visualizations (web, dashboards)

Feature	**Matplotlib**	**Plotly**
Performance	Handles large datasets efficiently	Can be slower for very large datasets (due to interactivity overhead)
3D Support	Basic 3D plotting (`mpl_toolkits.mplot3d`)	Strong 3D plotting (interactive 3D scatter, surface, mesh)

EXAMPLE - Matplotlib (Static Plot)

[ ]:
```
import matplotlib.pyplot as plt

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

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]
y = [10, 20, 25, 30]

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

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

Hierarchical indexing (also called a MultiIndex) in Pandas allows you to have multiple levels of row or column indexes in a DataFrame or Series.

Instead of just one row label, you can use two or more indexes, which creates a tree-like structure.

Why is it Significant:

1.Represents Higher-Dimensional Data in 2D

- Lets you work with higher-dimensional data (like 3D or 4D) in a 2D DataFrame format.

2.More Powerful Data Selection:

- You can access data using multiple keys (e.g., df.loc[('India', 'Delhi')]).

3.Better Data Organization:

- Useful for grouping and working with datasets that have natural hierarchical structures (e.g., Country → State → City).

4.Works Seamlessly with GroupBy

- Many groupby() operations return results with a MultiIndex.

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

Seaborn's pairplot() is a **high-level visualization tool used to quickly explore the relationships between multiple variables in a dataset**.

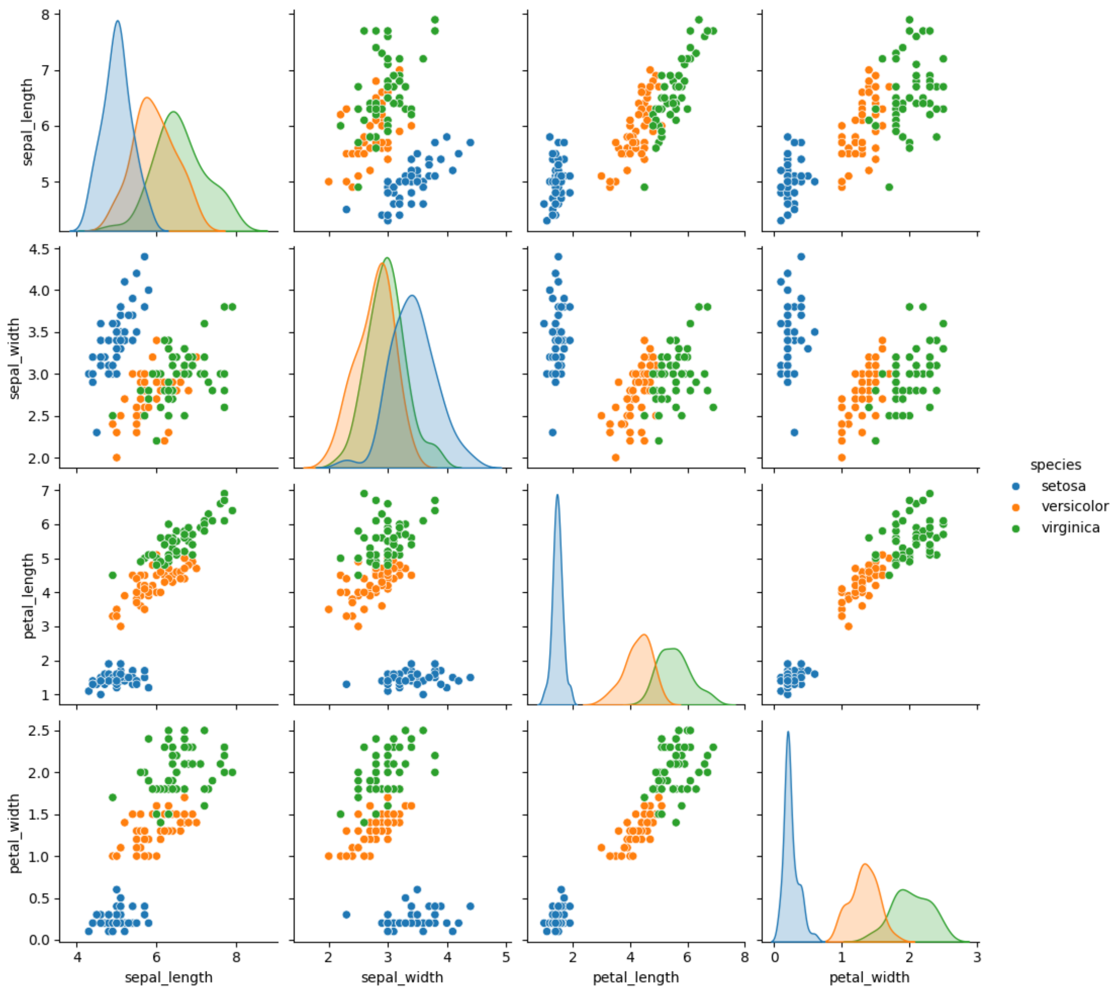
It creates a matrix of plots:

- **Diagonal** → Distribution of each variable (histogram or KDE).
- **Off-diagonal** → Scatter plots showing relationships between pairs of variables.

EXAMPLE:

```
[ ]: import seaborn as sns
import matplotlib.pyplot as plt

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

# Pairplot with hue for species
```

```
sns.pairplot(iris, hue="species", diag_kind="kde")
plt.show()
```



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

The describe() function in **Pandas is used to generate summary statistics** of a DataFrame (or Series). It gives you a quick overview of the distribution and key statistics of your data.

What It Returns

By default (for numerical columns), it provides:

- **count** → Number of non-null values
- **mean** → Average value
- **std** → Standard deviation (spread of values)
- **min** → Minimum value
- **25%** → First quartile (Q1)

- **50%** → Median (Q2)
- **75%** → Third quartile (Q3)
- **max** → Maximum value

EXAMPLE:

[ ]:
```python
import pandas as pd

data = {
    "Age": [22, 25, 29, 30, 32, 35],
    "Salary": [30000, 35000, 40000, 42000, 50000, 60000]
}

df = pd.DataFrame(data)

print(df.describe())
```

```
             Age        Salary
count   6.000000      6.000000
mean   28.833333  42833.333333
std     4.708149  10778.064143
min    22.000000  30000.000000
25%    26.000000  36250.000000
50%    29.500000  41000.000000
75%    31.500000  48000.000000
max    35.000000  60000.000000
```

Q13. Why is handling missing data important in Pandas?

1.Maintains Data Quality

- Missing values can reduce the **reliability and accuracy** of your analysis.
- For example, calculating an **Avarage** salary with missing entries might give misleading results.

2.Prevents Errors in Analysis

- Many Pandas/NumPy functions (like mean(), sum(), corr()) may return NaN if missing values are not handled.
- Machine learning models (like in scikit-learn) often cannot handle missing values directly.

3.Preserves Statistical Validity

- Missing data can bias results if not **treated properly**.
- **Example:** If younger people's ages are missing, the average age will be incorrectly higher.

4.Enables Better Modeling

- Models require complete, **clean data** to learn patterns correctly.
- Handling missing **data ensures models** don't fail or produce incorrect predictions.

5.Improves Decision-Making

- Clean datasets with no **unexpected gaps** lead to better business insights and **trustworthy reports**.

How Pandas Helps Handle Missing Data

- Pandas provides built-in functions like:
- df.isnull() → Detect missing values
- df.dropna() → Remove missing values
- df.fillna(value) → Replace missing values with a constant, mean, median, mode, etc.
- df.interpolate() → Estimate missing values using interpolation

EXAMPLE:

[ ]:
```python
import pandas as pd

data = {"Name": ["Alice", "Bob", "Charlie"],
        "Age": [25, None, 30],
        "Salary": [50000, 60000, None]}

df = pd.DataFrame(data)

print("Original Data:")
print(df)

# Fill missing Age with mean, Salary with 0
df["Age"].fillna(df["Age"].mean(), inplace=True)
df["Salary"].fillna(0, inplace=True)

print("\nAfter Handling Missing Data:")
print(df)
```

```
Original Data:
      Name   Age   Salary
0    Alice  25.0  50000.0
1      Bob   NaN  60000.0
2  Charlie  30.0      NaN

After Handling Missing Data:
      Name   Age   Salary
0    Alice  25.0  50000.0
1      Bob  27.5  60000.0
2  Charlie  30.0      0.0
```

```
/tmp/ipython-input-2156833814.py:13: FutureWarning: A value is trying to be set
on a copy of a DataFrame or Series through chained assignment using an inplace
method.
```

```
The behavior will change in pandas 3.0. This inplace method will never work
because the intermediate object on which we are setting values always behaves as
a copy.

For example, when doing 'df[col].method(value, inplace=True)', try using
'df.method({col: value}, inplace=True)' or df[col] = df[col].method(value)
instead, to perform the operation inplace on the original object.


  df["Age"].fillna(df["Age"].mean(), inplace=True)
/tmp/ipython-input-2156833814.py:14: FutureWarning: A value is trying to be set
on a copy of a DataFrame or Series through chained assignment using an inplace
method.
The behavior will change in pandas 3.0. This inplace method will never work
because the intermediate object on which we are setting values always behaves as
a copy.

For example, when doing 'df[col].method(value, inplace=True)', try using
'df.method({col: value}, inplace=True)' or df[col] = df[col].method(value)
instead, to perform the operation inplace on the original object.


  df["Salary"].fillna(0, inplace=True)
```

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

Benefits of Using Plotly for Data Visualization

1.Interactivity by Default

- Unlike Matplotlib/Seaborn (which are mostly static), **Plotly charts** are interactive.
- Features: zoom, pan, hover **tooltips, toggle legend items**, export as images.

2.Beautiful Visuals with Minimal Code

- Provides modern, **polished plots** without much customization.
- Example: px.line() creates a fully interactive line chart in one line.

3.Wide Range of Charts

- Supports simple charts (line, bar, scatter, pie) and advanced ones:
- 3D scatter, surface plots
- Choropleth (maps)
- Time-series charts
- Sankey diagrams, Treemaps, Sunbursts

4.Seamless Pandas & NumPy Integration

- Works directly with DataFrames → just pass column names **instead of manually extracting arrays.**

Q15.How does NumPy handle multidimensional arrays?

- **REPRESENTATION**

A NumPy array can have any number of dimensions:

- **1.1D** → vector ([1, 2, 3])
- **2.2D** → matrix ([[1, 2], [3, 4]])
- **3.3D** → tensor ([[[1,2],[3,4]], [[5,6],[7,8]]])

and so on…

The number of **dimensions** is called the rank of the array.

Dimensions themselves are referred to as **axes**.

EXAMPLE:

[1]:
```
import numpy as np

arr = np.array([[1, 2, 3],
                [4, 5, 6]])
print(arr.ndim)  # 2D array
print(arr.shape) # (2, 3) → 2 rows, 3 columns
```

```
2
(2, 3)
```

- **2.SHAPE & STRIDES**
- **Shape** → A tuple describing the size along each axis.
- **Example:** a 3×4×2 array has shape = (3, 4, 2)
- **Strides** → Tell NumPy how many bytes to step in each dimension when traversing.
- This makes slicing and reshaping very efficient without copying data.
- **3.INDEXING & SLICING**
- NumPy supports multidimensional indexing:

EXAMPLE:

[2]:
```
arr = np.array([[10, 20, 30],
                [40, 50, 60]])
print(arr[1, 2])   # 60
print(arr[:, 1])   # [20, 50] → second column
```

```
60
[20 50]
```

- **4.VECTORIZED OPERATIONS**:
- Operations are applied element-wise across dimensions:

EXAMPLE

[3]:
```python
A = np.array([[1, 2], [3, 4]])
B = np.array([[10, 20], [30, 40]])
print(A + B)  # element-wise addition
```

```
[[11 22]
 [33 44]]
```

- NumPy uses **broadcasting** rules to handle operations on arrays of different shapes.
- **5.RESHAPING & TRANSPOING**
- Arrays can be reshaped without changing data

EXAMPLE:

[4]:
```python
arr = np.arange(12).reshape(3, 4)
print(arr.T)   # transpose (swap axes)
```

```
[[ 0  4  8]
 [ 1  5  9]
 [ 2  6 10]
 [ 3  7 11]]
```

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

Role of Bokeh in Data Visualization

1.Interactive Visualizations

- Unlike static libraries (e.g., Matplotlib), Bokeh **emphasizes interactivity (zooming, panning, tooltips, filtering)**.
- **Example:** hover over a point to see details, or drag to **zoom into regions**.

2.Web-Friendly Output

- Bokeh generates **visualizations that can be rendered** in browsers using HTML, JavaScript, and JSON.
- It integrates seamlessly with **Flask, Django, or Jupyter Notebooks**.

3.Handling Large Datasets

- Bokeh is optimized to work with large or streaming datasets.
- It can use a Bokeh server to **stream and update data** in real time.

4.High-Level Charts & Customization

- Provides **high-level charting** (bar, line, scatter, heatmaps, etc.).
- Also allows low-level control to build custom, **complex dashboards**.

5.Integration with Other Tools

- Works well with **Pandas, NumPy, and Dask for data manipulation.**

- Can embed plots into **web apps or dashboards** (similar to Plotly Dash).

EXAMPLE:

[6]:
```
from bokeh.plotting import figure, show

# Create a simple line plot
p = figure(title="Simple Line Example", x_axis_label='x', y_axis_label='y')
p.line([1, 2, 3, 4], [6, 7, 2, 4], line_width=2)

show(p)  # Opens in a browser
```

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

Feature	`map()`	`apply()`
Works on	**Series only**	**Series & DataFrame**
Input types	Function, dictionary, Series	Function (any Python function, NumPy ufunc)
Output	Transformed Series	Series (if applied on Series) or DataFrame
Axis support	(not needed, always element-wise)	(`axis=0` for columns, `axis=1` for rows)

1. **map()**

- Works only on Series (one-dimensional).
- Applies a function, dictionary, or mapping to each element in the Series.
- Element-wise operation.

EXAMPLE:

[8]:
```
import pandas as pd

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

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

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

# Using a function on strings
s2 = pd.Series(['cat', 'dog', 'bat'])
print(s2.map(str.upper))
```

```
0     1
1     4
2     9
3    16
```

```
dtype: int64
0      A
1      B
2    NaN
3    NaN
dtype: object
0    CAT
1    DOG
2    BAT
dtype: object
```

2. **apply()**

- Works on both Series and DataFrame.
- On a Series → similar to map(), applies a function element-wise.
- On a DataFrame → applies a function along an axis (rows or columns).
- axis=0 → function applied column-wise
- axis=1 → function applied row-wise

EXAMPLE:

[9]:
```python
df = pd.DataFrame({
    'A': [1, 2, 3],
    'B': [10, 20, 30]
})

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

# Apply on DataFrame (column-wise sum)
print(df.apply(sum, axis=0))

# Apply on DataFrame (row-wise sum)
print(df.apply(sum, axis=1))
```

```
0    1
1    4
2    9
Name: A, dtype: int64
A     6
B    60
dtype: int64
0    11
1    22
2    33
dtype: int64
```

Q18.What are some advanced features of NumPy?

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

1. Broadcasting - Allows arithmetic operations between arrays of different shapes without explicit looping.

EXAMPLE:

[10]:
```
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
```

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

2. Vectorization

- Replaces Python loops with fast, low-level C implementations.
- Makes operations much faster than using for loops.

EXAMPLE:

[11]:
```
arr = np.array([10, 20, 30, 40, 50])
print(arr[[0, 3]])          # Fancy indexing → [10 40]
print(arr[arr > 25])        # Boolean indexing → [30 40 50]
```

```
[10 40]
[30 40 50]
```

3. Fancy Indexing & Boolean Indexing - Select elements using arrays of indices or conditions.

EXAMPLE:

[12]:
```
arr = np.array([10, 20, 30, 40, 50])
print(arr[[0, 3]])          # Fancy indexing → [10 40]
print(arr[arr > 25])        # Boolean indexing → [30 40 50]
```

```
[10 40]
[30 40 50]
```

4. Structured Arrays & Record Arrays

- Store heterogeneous data (like tables) in a single NumPy array.

EXAMPLE:

[13]:
```
data = np.array([(1, 'Alice', 3.5),
                 (2, 'Bob', 7.2)],
                dtype=[('id', 'i4'), ('name', 'U10'), ('score', 'f4')])
print(data['name'])
```

```
['Alice' 'Bob']
```

5. **Universal Functions** (ufuncs)

- Highly optimized element-wise functions (e.g., np.sin, np.exp, np.add).
- Support broadcasting and can be combined with reduce, accumulate, outer.

EXAMPLE:

[14]:
```python
x = np.array([1, 2, 3])
print(np.add.reduce(x))  # sum → 6
```

```
6
```

Q19.How does Pandas simplify time series analysis? Pandas was originally built with time **series analysis in mind**, so it provides a lot of **tools** that make working with dates, times, and indexed data much simpler compared to plain Python.

1. **Date and Time Indexing**

- Pandas has a special DatetimeIndex that lets you use dates and times as the index.
- This allows label-based indexing with time stamps.

[15]:
```python
import pandas as pd

# Create a time series
dates = pd.date_range("2025-01-01", periods=5, freq="D")
ts = pd.Series([10, 20, 30, 40, 50], index=dates)

print(ts["2025-01-03"])   # Access by date → 30
print(ts["2025-01"])      # Slice by month
```

```
30
2025-01-01    10
2025-01-02    20
2025-01-03    30
2025-01-04    40
2025-01-05    50
Freq: D, dtype: int64
```

2. **Resampling and Frequency Conversion**

- Easily convert data between frequencies (e.g., daily → monthly, hourly → daily).
- Supports aggregation (mean, sum, etc.) and upsampling/downsampling.

[16]:
```python
print(ts.resample("M").mean())  # Monthly average
```

```
2025-01-31    30.0
Freq: ME, dtype: float64
```

```
/tmp/ipython-input-1560530645.py:1: FutureWarning: 'M' is deprecated and will be
removed in a future version, please use 'ME' instead.
  print(ts.resample("M").mean())  # Monthly average
```

3. **Shifting and Lagging**

   - You can shift data forward or backward in time to calculate changes, returns, etc.

[17]:
```python
print(ts.shift(1))     # Lagged values
print(ts.diff())       # First difference
```

```
2025-01-01     NaN
2025-01-02    10.0
2025-01-03    20.0
2025-01-04    30.0
2025-01-05    40.0
Freq: D, dtype: float64
2025-01-01     NaN
2025-01-02    10.0
2025-01-03    10.0
2025-01-04    10.0
2025-01-05    10.0
Freq: D, dtype: float64
```

4. **Rolling, Expanding, and Moving Windows**

   - Built-in support for rolling statistics, moving averages, expanding windows.

[18]:
```python
print(ts.rolling(window=3).mean())  # 3-day moving average
```

```
2025-01-01     NaN
2025-01-02     NaN
2025-01-03    20.0
2025-01-04    30.0
2025-01-05    40.0
Freq: D, dtype: float64
```

5. **Time Zone Handling**

   - Pandas supports time zone–aware DatetimeIndex objects and conversion.

[19]:
```python
ts_utc = ts.tz_localize("UTC")
print(ts_utc.tz_convert("Asia/Kolkata"))
```

```
2025-01-01 05:30:00+05:30    10
2025-01-02 05:30:00+05:30    20
2025-01-03 05:30:00+05:30    30
2025-01-04 05:30:00+05:30    40
2025-01-05 05:30:00+05:30    50
Freq: D, dtype: int64
```

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

In Pandas, a pivot table plays the role of a powerful tool for summarizing, aggregating, and reorganizing data — very similar to pivot tables in Excel.

Role of a Pivot Table in Pandas

1.Data Summarization

- Pivot tables group data by one or more keys (rows/columns) and apply an aggregation function (like mean, sum, count, etc.).
- This helps in quickly getting insights from large datasets.

2.Reshaping Data

- Transforms data from long format to wide format.
- Useful for cross-tabulations and comparisons between categories.

3.Aggregation and Statistics

- Supports multiple aggregation functions (mean, sum, min, max, count).
- Can show multiple aggregations at once.

4.Multi-level Grouping

- Allows grouping by multiple columns (hierarchical indexing).
- Helps analyze data across several dimensions.

EXAMPLE:

[20]:
```python
import pandas as pd

# Sample dataset
data = {
    'Department': ['Sales', 'Sales', 'HR', 'HR', 'IT', 'IT'],
    'Employee': ['A', 'B', 'C', 'D', 'E', 'F'],
    'Salary': [5000, 6000, 4500, 4800, 7000, 7200],
    'Bonus': [500, 600, 300, 400, 800, 900]
}
df = pd.DataFrame(data)

# Create a pivot table
pivot = pd.pivot_table(df,
                       values=['Salary', 'Bonus'],
                       index='Department',
                       aggfunc={'Salary': 'mean', 'Bonus': 'sum'})

print(pivot)
```

```
            Bonus  Salary
Department
HR            700  4650.0
```

```
IT            1700  7100.0
Sales         1100  5500.0
```

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

NumPy's array slicing is much faster than **Python's built-in list slicing**, and the main reasons come from how they are implemented under the hood.

1. **Memory Layout**

- **Python lists:**
- A list is an array of **pointers to objects scattered** in memory.
- **Slicing a list creates** a new list with copies of references (extra overhead).
- **NumPy arrays:**
- Stored as a **contiguous block of memory** (like a C array).
- Slicing does not **copy data; instead, it creates a view into the same memory buffer using strides**.
- This makes slicing constant-time (no data copying).

2. **Views vs Copies**

- **Python list slicing** → Always makes a new object (copy of references).
- **NumPy array slicing** → Returns a view (unless explicitly copied).

[21]:
```python
import numpy as np

arr = np.arange(10)
slice_arr = arr[2:6]    # view, no data copied
slice_arr[0] = 99
print(arr)              # original array also changes
```

```
[ 0  1 99  3  4  5  6  7  8  9]
```

3. **Vectorized Implementation**

- NumPy is built on C and Fortran libraries (BLAS/LAPACK).
- Slicing just adjusts pointers and strides instead of looping over elements.
- Python **lists, being high-level,** need to loop element by element in pure Python → slower.

4. **Data Type Uniformity**

- NumPy arrays have a single data type (dtype), which allows efficient pointer arithmetic.
- Python lists can contain mixed types, so slicing needs type checks and extra overhead.

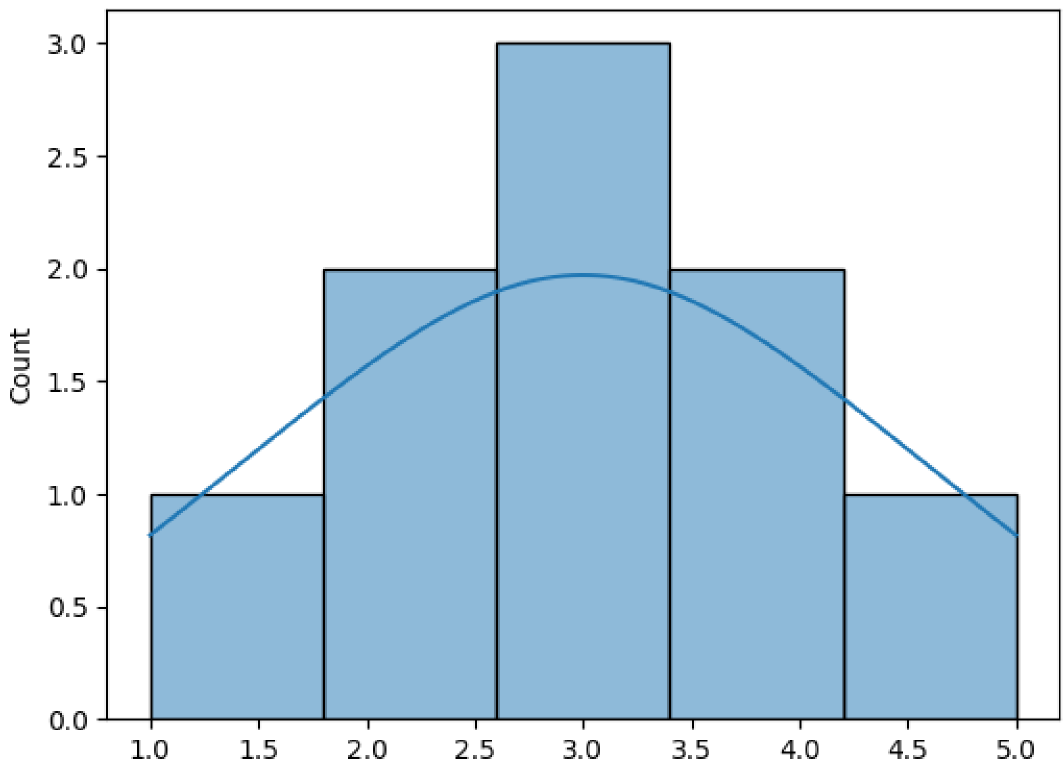
Q22.What are some common use cases for Seaborn?

Seaborn is a Python data visualization library built on top of Matplotlib, designed for making statistical **graphics easier and prettier**. It's widely used in data analysis and machine learning workflows.

1. **Exploratory Data Analysis** (EDA)
   - Quickly understand **distributions, relationships, and patterns** in datasets.
   - **Example:** visualizing the spread of numerical variables or comparing categories.
2. **Visualizing Distributions**
   - Functions like histplot(), kdeplot(), distplot() (deprecated) show probability distributions.
   - Useful for detecting skewness, outliers, and spread of data.

[22]:
```python
import seaborn as sns
sns.histplot(data=[1,2,2,3,3,3,4,4,5], kde=True)
```

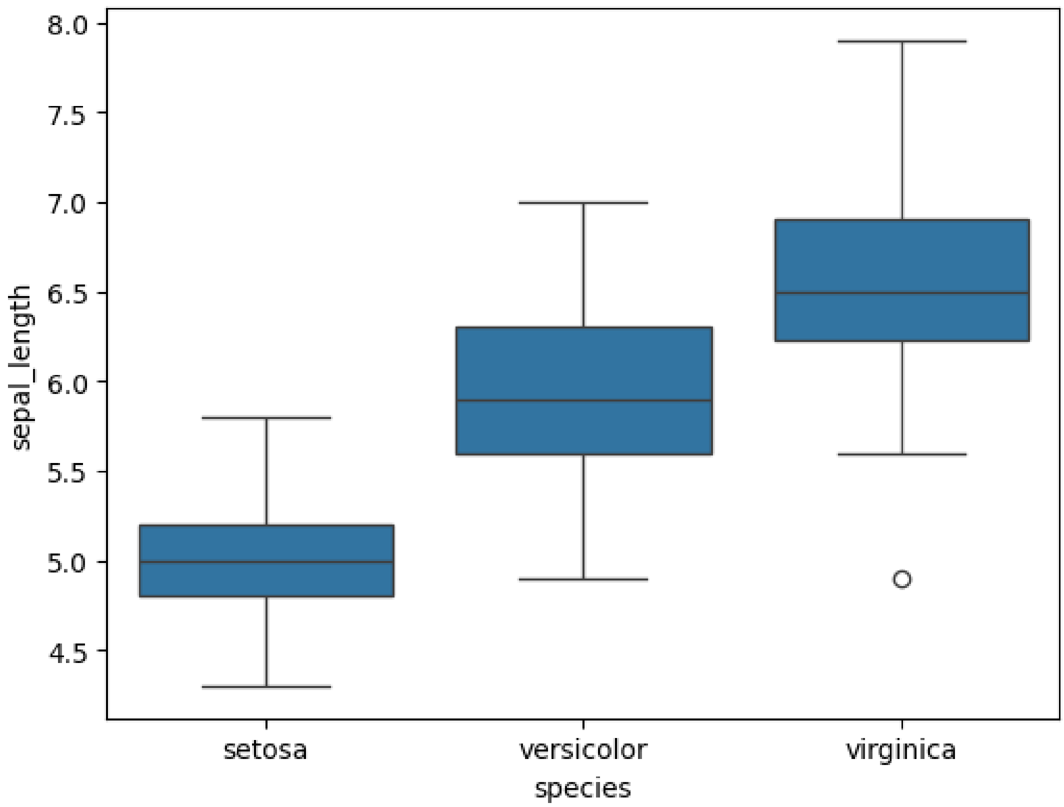
[22]:
```
<Axes: ylabel='Count'>
```



3. **Comparing Categories**
   - Bar plots, count plots, and box plots for categorical data analysis.
   - Great for understanding differences between groups

[23]:
```python
sns.boxplot(x="species", y="sepal_length", data=sns.load_dataset("iris"))
```

[23]:
```
<Axes: xlabel='species', ylabel='sepal_length'>
```

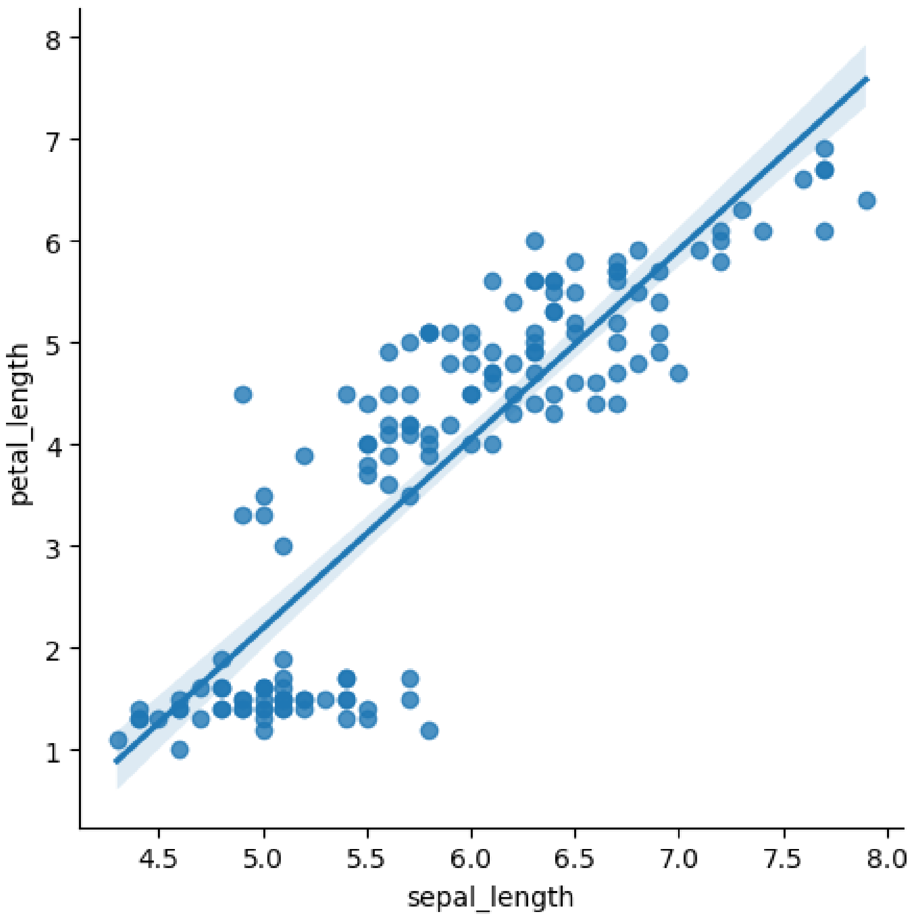


4. **Relationship Analysis**

   - Scatter plots, regression lines (lmplot, regplot) to analyze correlation between variables.

[24]:
```
sns.lmplot(x="sepal_length", y="petal_length", data=sns.load_dataset("iris"))
```

[24]: <seaborn.axisgrid.FacetGrid at 0x790e962959a0>

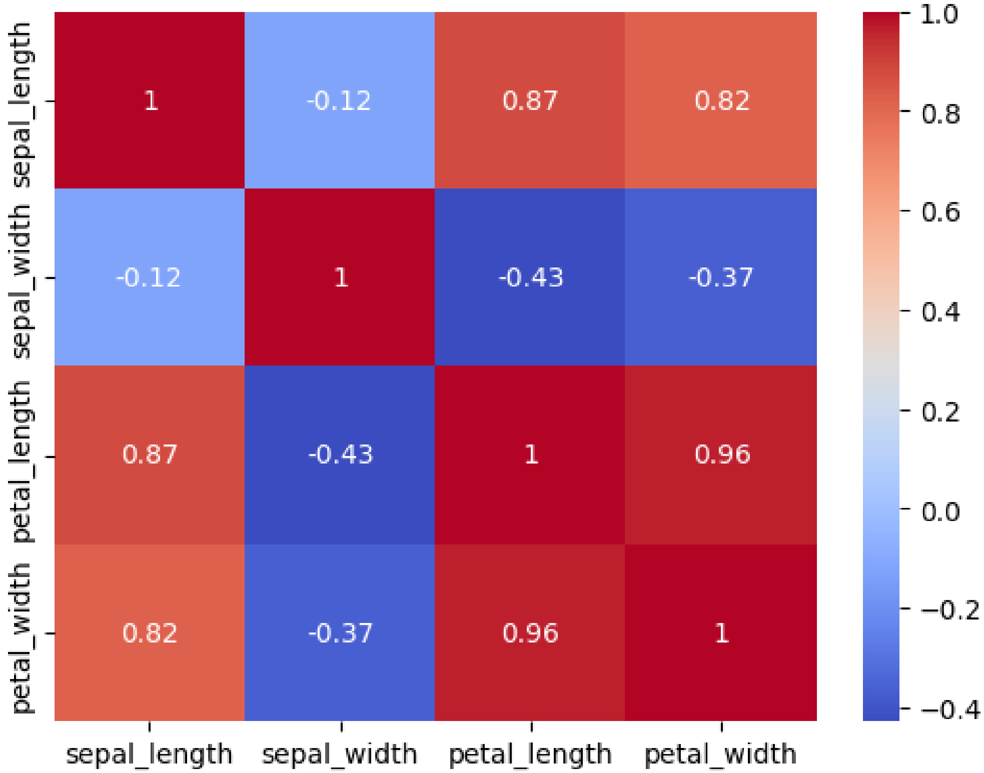


5. Heatmaps & Correlation Analysis

- heatmap() is often used to visualize correlation matrices in feature analysis

[26]:
```
import pandas as pd
iris = sns.load_dataset("iris")
# Drop the non-numerical 'species' column before calculating correlation
sns.heatmap(iris.drop('species', axis=1).corr(), annot=True, cmap="coolwarm")
```

[26]:
```
<Axes: >
```



2 PRACTICAL

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

[27]:
```
import numpy as np

# Create a 2D NumPy array
array_2d = np.array([[1, 2, 3],
                     [4, 5, 6],
                     [7, 8, 9]])

print("2D NumPy Array:")
print(array_2d)

# Calculate the sum of each row
# The axis=1 argument specifies that the sum should be calculated across␣
↪columns (for each row)
row_sums = np.sum(array_2d, axis=1)

print("\nSum of each row:")
print(row_sums)
```

```
2D NumPy Array:
[[1 2 3]
 [4 5 6]
 [7 8 9]]

Sum of each row:
[ 6 15 24]
```

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

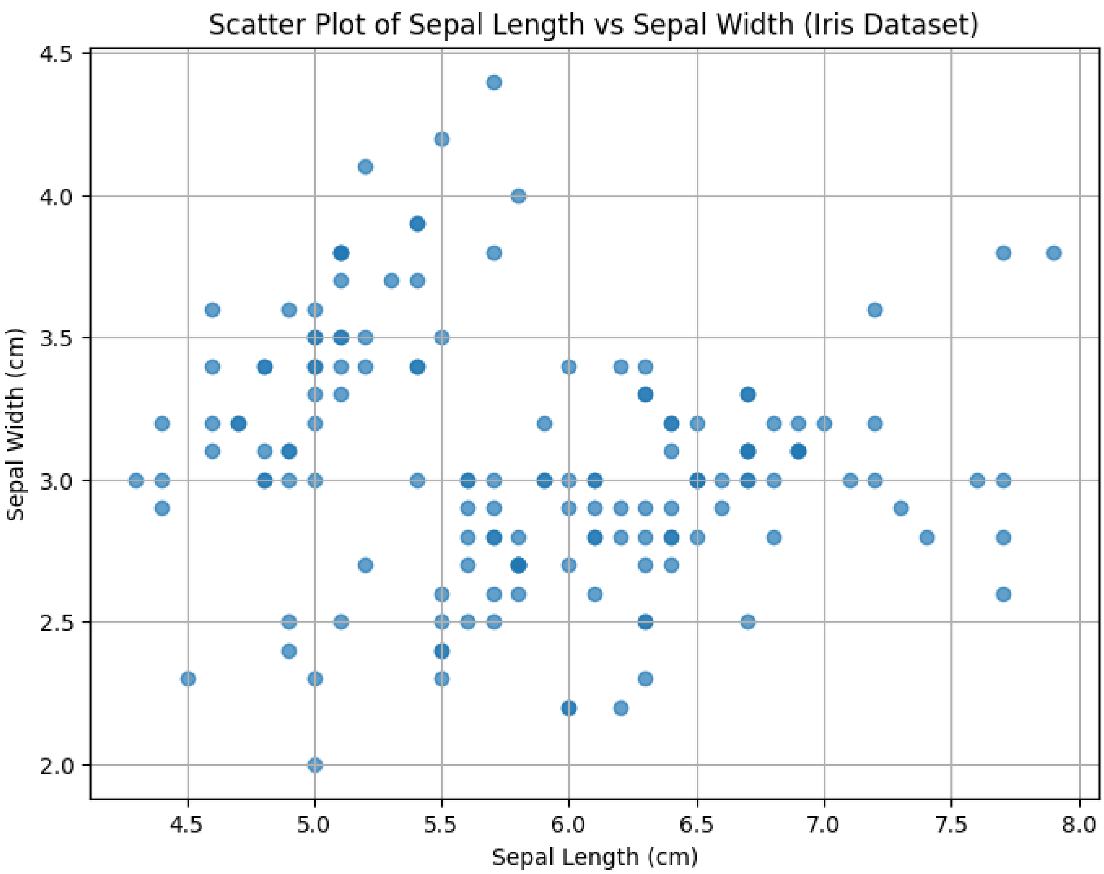
[ ]:

```python
[28]: # Find the mean of a specific column (e.g., 'Salary')
mean_salary = df['Salary'].mean()

print(f"The mean salary is: {mean_salary}")
```

```
The mean salary is: 5750.0
```

Q3.Create a scatter plot using Matplotlib?

```python
[29]: import matplotlib.pyplot as plt
import seaborn as sns # Often used with matplotlib for datasets and styling

# Load the iris dataset if not already loaded (though it is in this notebook)
# iris = sns.load_dataset("iris")

# Create a scatter plot
plt.figure(figsize=(8, 6)) # Optional: set figure size
plt.scatter(iris['sepal_length'], iris['sepal_width'], alpha=0.7) # alpha for␣
↪transparency
plt.title('Scatter Plot of Sepal Length vs Sepal Width (Iris Dataset)')
plt.xlabel('Sepal Length (cm)')
plt.ylabel('Sepal Width (cm)')
plt.grid(True) # Optional: add a grid
plt.show()
```

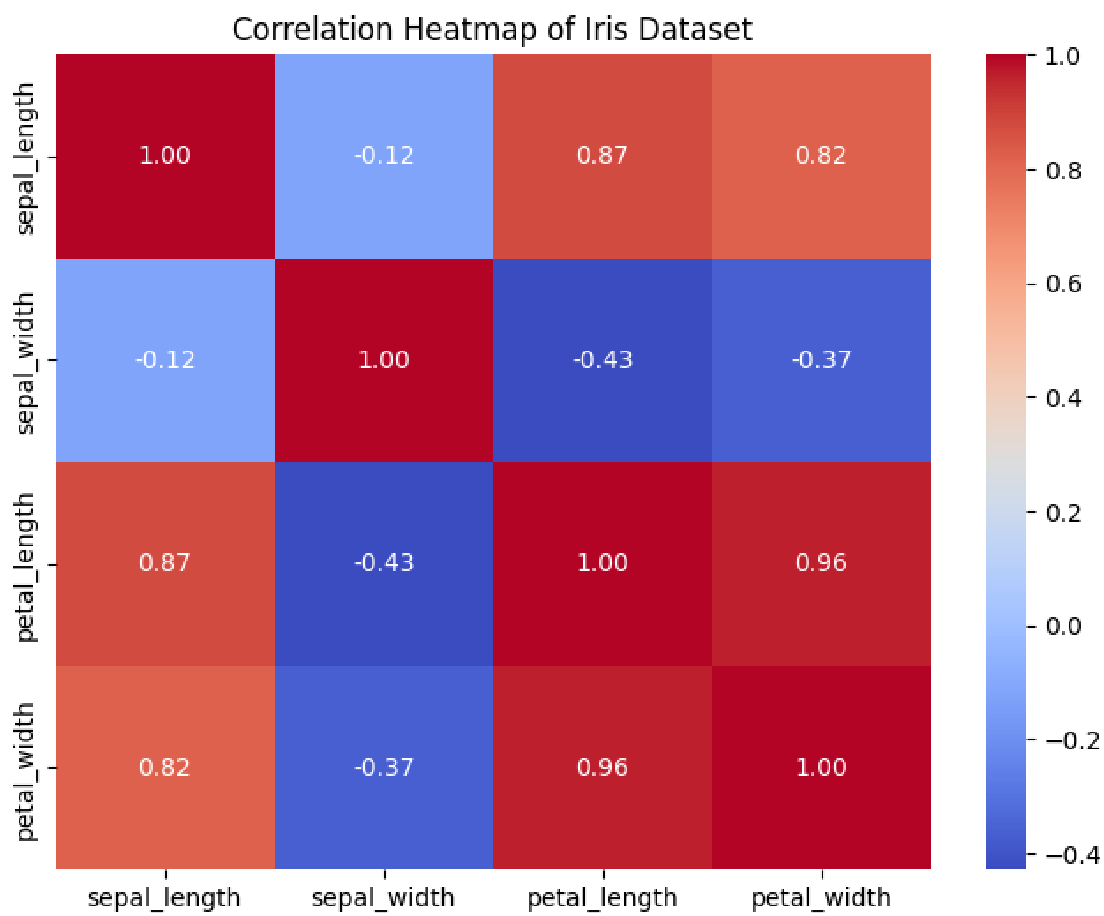


Q4.How do you calculate the correlation matrix using Seaborn and visualize it with a heatmap?

[30]:
```
import seaborn as sns
import matplotlib.pyplot as plt
import pandas as pd

# Load the iris dataset (if not already loaded)
iris = sns.load_dataset("iris")

# Calculate the correlation matrix, excluding the non-numerical 'species' column
correlation_matrix = iris.drop('species', axis=1).corr()

print("Correlation Matrix:")
print(correlation_matrix)

# Visualize the correlation matrix using a heatmap
plt.figure(figsize=(8, 6)) # Optional: set figure size
sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm', fmt=".2f") # annot␣
↪shows values, fmt formats them
```

```
plt.title('Correlation Heatmap of Iris Dataset')
plt.show()
```

```
Correlation Matrix:
              sepal_length  sepal_width  petal_length  petal_width
sepal_length      1.000000    -0.117570      0.871754     0.817941
sepal_width      -0.117570     1.000000     -0.428440    -0.366126
petal_length      0.871754    -0.428440      1.000000     0.962865
petal_width       0.817941    -0.366126      0.962865     1.000000
```



Q5.Generate a bar plot using Plotly?

[31]:
```
import plotly.express as px

# Use the 'pivot' DataFrame created earlier for department-wise sums
# Or use the original df and group by department to calculate the sum
# Here, I'll use the original df and group by department to get the sum of␣
↪salaries
department_salary_sum = df.groupby('Department')['Salary'].sum().reset_index()
```

```python
# Create a bar plot using Plotly Express
fig = px.bar(department_salary_sum,
             x='Department',
             y='Salary',
             title='Total Salary by Department')

fig.show()
```

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

[32]:
```python
import pandas as pd

# Create a DataFrame
data = {'Numbers': [1, 2, 3, 4, 5]}
df_new = pd.DataFrame(data)

print("Original DataFrame:")
print(df_new)

# Add a new column 'Squared' based on the 'Numbers' column
df_new['Squared'] = df_new['Numbers'] ** 2

print("\nDataFrame with new column:")
print(df_new)
```

```
Original DataFrame:
   Numbers
0        1
1        2
2        3
3        4
4        5

DataFrame with new column:
   Numbers  Squared
0        1        1
1        2        4
2        3        9
3        4       16
4        5       25
```

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

[34]:
```python
import numpy as np

# Create two NumPy arrays
array1 = np.array([[1, 2],
```

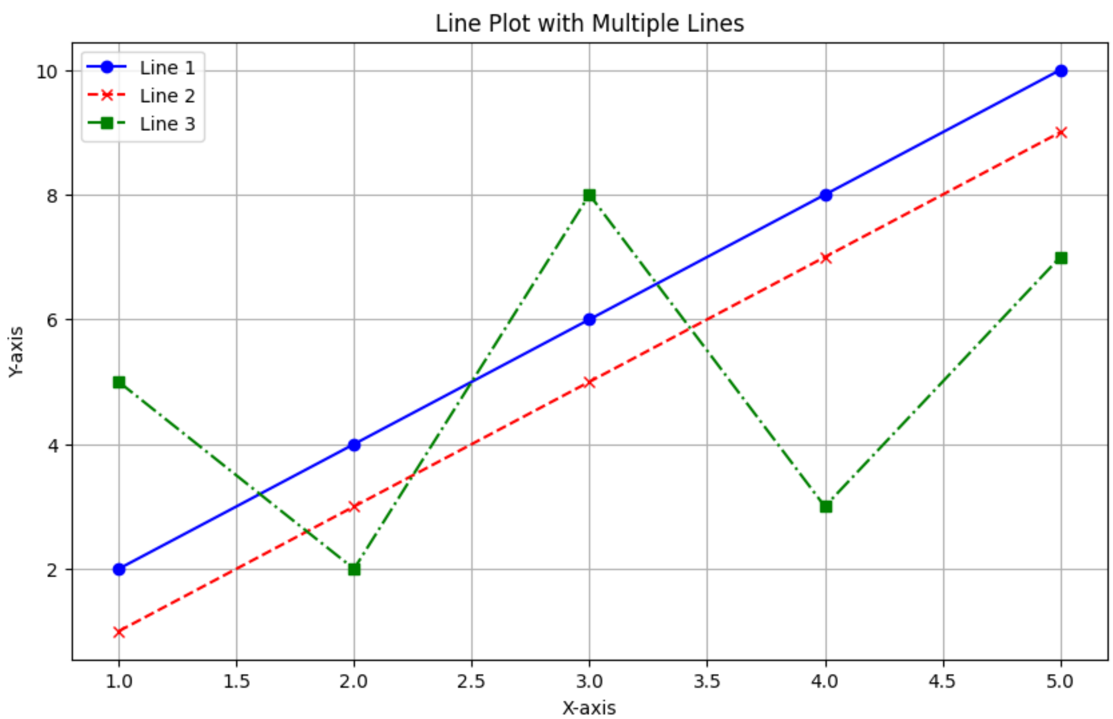
```
                   [3, 4]])

array2 = np.array([[5, 6],
                   [7, 8]])

print("Array 1:")
print(array1)

print("\nArray 2:")
print(array2)

# Perform element-wise multiplication
result_array = array1 * array2

print("\nResult of element-wise multiplication:")
print(result_array)
```

```
Array 1:
[[1 2]
 [3 4]]

Array 2:
[[5 6]
 [7 8]]

Result of element-wise multiplication:
[[ 5 12]
 [21 32]]
```

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

[35]:
```
import matplotlib.pyplot as plt
import numpy as np

# Sample data
x = np.array([1, 2, 3, 4, 5])
y1 = np.array([2, 4, 6, 8, 10])
y2 = np.array([1, 3, 5, 7, 9])
y3 = np.array([5, 2, 8, 3, 7])

# Create the plot
plt.figure(figsize=(10, 6)) # Optional: set figure size

plt.plot(x, y1, marker='o', linestyle='-', color='blue', label='Line 1')
plt.plot(x, y2, marker='x', linestyle='--', color='red', label='Line 2')
plt.plot(x, y3, marker='s', linestyle='-.', color='green', label='Line 3')

plt.title('Line Plot with Multiple Lines')
```

```python
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.legend() # Show the legend to identify lines
plt.grid(True)
plt.show()
```



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

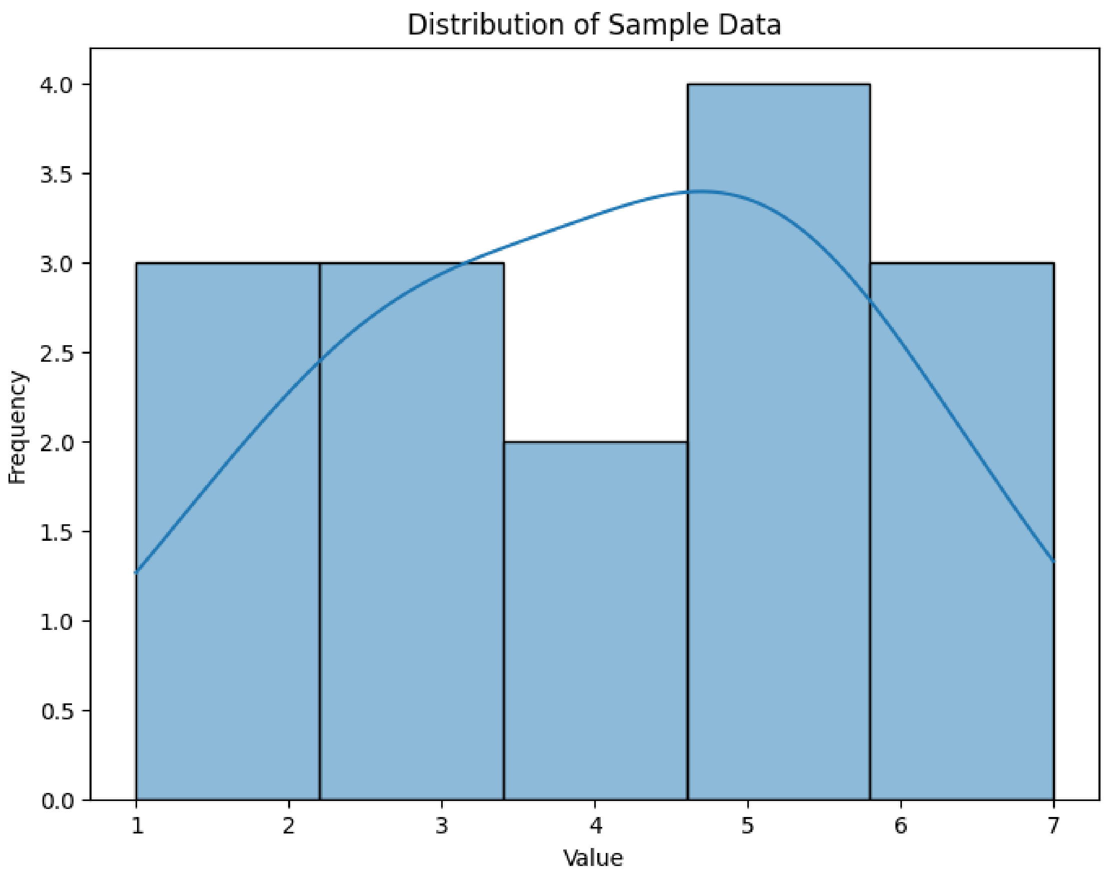
[36]:
```python
import pandas as pd

# Create a sample DataFrame
data = {'Numbers': [10, 25, 5, 40, 15, 30]}
df_filter = pd.DataFrame(data)

print("Original DataFrame:")
print(df_filter)

# Define a threshold
threshold = 20

# Filter rows where the 'Numbers' column is greater than the threshold
filtered_df = df_filter[df_filter['Numbers'] > threshold]
```

```python
print(f"\nDataFrame filtered for 'Numbers' > {threshold}:")
print(filtered_df)
```

```
Original DataFrame:
   Numbers
0       10
1       25
2        5
3       40
4       15
5       30

DataFrame filtered for 'Numbers' > 20:
   Numbers
1       25
3       40
5       30
```

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

[37]:
```python
import seaborn as sns
import matplotlib.pyplot as plt
import numpy as np

# Create some sample data for the histogram
data_for_histogram = [1, 2, 2, 3, 3, 3, 4, 4, 5, 5, 5, 5, 6, 6, 7]

# Create the histogram using Seaborn's histplot
plt.figure(figsize=(8, 6)) # Optional: set figure size
sns.histplot(data=data_for_histogram, kde=True) # kde=True adds a kernel␣
 ↪density estimate line
plt.title('Distribution of Sample Data')
plt.xlabel('Value')
plt.ylabel('Frequency')
plt.show()
```



Q11.Perform matrix multiplication using NumPy?

[38]:
```python
import numpy as np

# Create two NumPy matrices (2D arrays)
matrix1 = np.array([[1, 2],
                    [3, 4]])

matrix2 = np.array([[5, 6],
                    [7, 8]])

print("Matrix 1:")
print(matrix1)

print("\nMatrix 2:")
print(matrix2)

# Perform matrix multiplication using the @ operator (preferred in Python 3.5+)
result_matrix = matrix1 @ matrix2
```

```python
# Alternatively, using np.dot()
# result_matrix = np.dot(matrix1, matrix2)

print("\nResult of matrix multiplication:")
print(result_matrix)
```

```
Matrix 1:
[[1 2]
 [3 4]]

Matrix 2:
[[5 6]
 [7 8]]

Result of matrix multiplication:
[[19 22]
 [43 50]]
```

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

[39]:
```python
import pandas as pd

# Replace 'sample.csv' with the actual path to your CSV file
try:
    df_csv = pd.read_csv('sample.csv')

    print("DataFrame loaded from CSV:")
    # Display the first 5 rows
    display(df_csv.head())

except FileNotFoundError:
    print("Error: 'sample.csv' not found. Please replace 'sample.csv' with the␣
 ↪correct path to your file.")
except Exception as e:
    print(f"An error occurred: {e}")
```

```
Error: 'sample.csv' not found. Please replace 'sample.csv' with the correct path
to your file.
```

Q13.Create a 3D scatter plot using Plotly?

[40]:
```python
import plotly.express as px
import seaborn as sns

# Load the iris dataset if not already loaded
iris = sns.load_dataset("iris")

# Create a 3D scatter plot
```

```
fig = px.scatter_3d(iris,
                    x='sepal_length',
                    y='sepal_width',
                    z='petal_length',
                    color='species',
                    title='3D Scatter Plot of Iris Dataset')

fig.show()
```