PRACTICAL NO.1

Aim: To install the Matplotlib library in Python and use it to create basic plots such as line plot, bar chart, scatter plot, and histogram.

Theory:

Matplotlib is a widely used plotting library in Python, primarily used for 2D visualizations. It enables users to generate plots, histograms, power spectra, bar charts, error charts, and more with just a few lines of code.

- The matplotlib.pyplot module provides a MATLAB-like interface.
- Plots are highly customizable (titles, labels, legends, colors, etc.)
- Matplotlib supports exporting graphics to different file formats (PNG, PDF, SVG, etc.)

Key Features:

- Easy to use for beginners.
- Works well with NumPy and pandas.
- Produces publication-quality figures.

Code:

Step 1: Install Matplotlib

```
pip install matplotlib
```

Step 2: Basic Plotting Examples

```
import matplotlib.pyplot as plt
import numpy as np

# 1. Line Plot
x = [1, 2, 3, 4, 5]
y = [2, 3, 5, 7, 11]

plt.plot(x, y, marker='o')
plt.title("Line Plot")
plt.xlabel("X-axis")
plt.ylabel("Y-axis")
plt.grid(True)
plt.show()

# 2. Bar Chart
categories = ['A', 'B', 'C', 'D']
values = [4, 7, 1, 8]
plt.bar(categories, values, color='orange')
plt.title("Bar Chart")
plt.xlabel("Category")
plt.ylabel("Value")
plt.show()
```

```
# 3. Scatter Plot
x = [1, 2, 3, 4, 5]
y = [5, 7, 4, 6, 5]

plt.scatter(x, y, color='red')
plt.title("Scatter Plot")
plt.xlabel("X-axis")
plt.ylabel("Y-axis")
plt.show()


# 4. Histogram
data = np.random.randn(1000)

plt.hist(data, bins=20, color='green')
plt.title("Histogram")
plt.xlabel("Value")
plt.ylabel("Frequency")
plt.show()
```

Expected Output: A circular pie chart showing percentage distribution.

PRACTICAL NO: 2

Aim:

To install and use the Seaborn library for creating visually appealing and informative statistical graphics in Python.

Theory:

Seaborn is a Python data visualization library based on Matplotlib. It provides a high-level interface for drawing attractive and informative statistical graphics.

It works well with pandas DataFrames and integrates closely with NumPy and SciPy.

Key Features:

- Built-in themes and color palettes.
- Easy integration with pandas DataFrames.
- Functions for visualizing univariate and bivariate distributions.
- Automatic estimation and plotting of linear regression models.

✸ Common Seaborn Plots:

- sns.histplot() – histogram with optional KDE.
- sns.scatterplot() – scatter plot.
- sns.boxplot() – box and whisker plot.
- sns.barplot() – bar chart with confidence intervals.
- sns.pairplot() – multiple pairwise relationships.

Code:

☑ Step 1: Install Seaborn

```
pip install seaborn
```

☑ Step 2: Python Program Using Seaborn

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

# Load built-in dataset
tips = sns.load_dataset("tips")
```

```
# 1. Histogram
sns.histplot(data=tips, x="total_bill", kde=True)
plt.title("Histogram of Total Bill")
plt.show()


# 2. Scatter Plot
sns.scatterplot(data=tips, x="total_bill", y="tip", hue="sex")
plt.title("Total Bill vs Tip by Gender")
plt.show()


# 3. Box Plot
sns.boxplot(data=tips, x="day", y="total_bill", hue="sex")
plt.title("Boxplot of Total Bill by Day and Gender")
plt.show()


# 4. Pair Plot
sns.pairplot(tips, hue="sex")
plt.suptitle("Pairplot of Tips Dataset", y=1.02)
plt.show()
```

Expected Output:

1. A histogram showing the distribution of total bill values with a smooth KDE line.
2. A scatter plot showing the relationship between total bill and tip, color-coded by gender.
3. A box plot comparing total bill amounts across different days, with male/female differentiation.
4. A multi-panel pairplot showing pairwise relationships among numerical variables (e.g., total_bill, tip, size), color-coded by gender.

PRACTICAL NO. 3

Aim:

To compare Matplotlib, Seaborn, and Plotly libraries in Python for creating visualizations and to understand their strengths, limitations, and best use cases based on different data types and visualization needs.

Theory:

Data visualization is a crucial step in data analysis, and Python offers several libraries for this. Among the most popular are Matplotlib, Seaborn, and Plotly. Each serves different purposes and has unique features:

Feature	Matplotlib	Seaborn	Plotly
Type	Low-level	High-level (built on Matplotlib)	High-level & interactive
Ease of use	Requires more code	Easier with built-in themes	Very intuitive with interactivity
Interactivity	Static	Static	Interactive (zoom, hover, etc.)
Customization	Highly customizable	Limited to Matplotlib backend	Very customizable
Best use case	Custom static plots	Statistical data visualization	Web-based, interactive dashboards

Code:

☑ **Step 1: Install Required Libraries**

```
pip install matplotlib seaborn plotly
```

☑ **Step 2: Comparison Code**

```
# Import libraries
import matplotlib.pyplot as plt
import seaborn as sns
import plotly.express as px
import pandas as pd


# Load dataset
tips = sns.load_dataset("tips")
```

```
# ---------------- Matplotlib ----------------
plt.figure(figsize=(6, 4))
plt.scatter(tips['total_bill'], tips['tip'], c='blue')
plt.title("Matplotlib: Total Bill vs Tip")
plt.xlabel("Total Bill")
plt.ylabel("Tip")
plt.grid(True)
plt.show()

# ---------------- Seaborn ----------------
sns.scatterplot(data=tips, x='total_bill', y='tip', hue='sex')
plt.title("Seaborn: Total Bill vs Tip by Gender")
plt.show()

# ---------------- Plotly ----------------
fig = px.scatter(
    tips,
    x='total_bill',
    y='tip',
    color='sex',
    title='Plotly: Total Bill vs Tip by Gender',
    labels={'total_bill': 'Total Bill', 'tip': 'Tip'}
)
fig.show()
```

Expected Output:

- Matplotlib Plot: A basic, static scatter plot with blue dots, no interactivity.
- Seaborn Plot: A more polished static plot with gender-based coloring, legend, and better aesthetics.
- Plotly Plot: A fully interactive scatter plot (hover, zoom, pan), also with gender-based color.

PRACTICAL NO. 4

Aim:

To load a dataset using the Pandas library in Python and perform basic data cleaning tasks such as handling missing values and duplicate entries.

Theory:

Pandas is a powerful Python library used for data manipulation and analysis. It provides two primary data structures:

- Series – 1D labeled array.
- DataFrame – 2D table of data (rows and columns), similar to an Excel spreadsheet.

Data Cleaning is a crucial step before analysis or modeling. Common tasks include:

- Handling Missing Values:
  - df.isnull() – identifies missing values.
  - df.dropna() – removes rows with missing values.
  - df.fillna(value) – fills missing values with a given value.
- Handling Duplicates:
  - df.duplicated() – returns boolean Series for duplicate rows.
  - df.drop_duplicates() – removes duplicate rows.

Code:

☑ Step 1: Import Library

```
import pandas as pd
```

☑ Step 2: Create a Sample Dataset (with Missing Values & Duplicates)

```
# Sample dataset
data = {
    'Name': ['Alice', 'Bob', 'Charlie', 'Alice', None],
    'Age': [25, 30, None, 25, 22],
    'City': ['New York', 'Los Angeles', 'Chicago', 'New York', 'Miami']
}

# Load data into a DataFrame
df = pd.DataFrame(data)

print("Original Dataset:")
print(df)
```

☑ Step 3: Handling Missing Values

```
print("\nMissing Values in Dataset:")
print(df.isnull().sum())

# Fill missing 'Age' with mean and drop missing 'Name'
df['Age'].fillna(df['Age'].mean(), inplace=True)
```

```
df.dropna(subset=['Name'], inplace=True)
```

☑ Step 4: Handling Duplicates

```
print("\nCheck for Duplicate Rows:")
print(df.duplicated())

# Drop duplicate rows
df.drop_duplicates(inplace=True)
```

☑ Step 5: Display the Final Cleaned Dataset

```
print("\nCleaned Dataset:")
print(df)
```

Expected Output:

```
Original Dataset:
      Name   Age         City
0     Alice  25.0     New York
1       Bob  30.0  Los Angeles
2   Charlie   NaN      Chicago
3     Alice  25.0     New York
4      None  22.0        Miami

Missing Values in Dataset:
Name    1
Age     1 City    0
dtype: int64

Check for Duplicate Rows:
0    False
1    False
2    False
3     True
Name: duplicated, dtype: bool

Cleaned Dataset:
      Name        Age         City
0     Alice  25.000000     New York
1       Bob  30.000000  Los Angeles
2   Charlie  26.666667      Chicago
```

Note: The missing Age is filled with the average (≈ 26.67), and duplicate rows and missing names are removed.

PRACTICAL NO: 5

Aim:

To preprocess categorical and numerical data using Pandas and NumPy for effective visualization and analysis.

Theory:

In real-world datasets, raw data often needs preprocessing to be suitable for visualization or machine learning.

Preprocessing includes:

- Handling missing values
- Converting categorical data to numerical form
- Scaling or transforming numerical data

◇ Categorical Data:

- These are data with limited, fixed values (e.g., gender, city).
- Often stored as text and need encoding for numeric analysis.
- Techniques:
  - ○ Label Encoding – assigns each unique category a number.
  - ○ One-Hot Encoding – creates binary columns for each category.

◇ Numerical Data:

- Continuous or discrete numbers (e.g., age, income).
- May need normalization or scaling.
- Can be processed using NumPy for calculations (mean, std, etc.)

Code:

☑ Step 1: Import Required Libraries

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

☑ Step 2: Create a Sample Dataset

```
# Sample dataset with categorical and numerical data
data = {
    'Name': ['Alice', 'Bob', 'Charlie', 'David', 'Eva'],
    'Gender': ['Female', 'Male', 'Male', 'Male', 'Female'],
    'Age': [25, 30, np.nan, 22, 28],
    'City': ['New York', 'Los Angeles', 'Chicago', 'Chicago', 'Miami']
}

# Create DataFrame
df = pd.DataFrame(data)
print("Original Data:")
print(df)
```

☑ Step 3: Handle Missing Numerical Values

```
# Fill missing 'Age' with the mean value
df['Age'].fillna(df['Age'].mean(), inplace=True)
```

☑ Step 4: Encode Categorical Columns

```
# Label Encoding for 'Gender'
df['Gender_encoded'] = df['Gender'].map({'Female': 0, 'Male': 1})

# One-Hot Encoding for 'City'
city_dummies = pd.get_dummies(df['City'], prefix='City')

# Merge one-hot encoded columns with the main DataFrame
df = pd.concat([df, city_dummies], axis=1)

# Drop original categorical columns
df.drop(['Gender', 'City'], axis=1, inplace=True)
```

☑ Step 5: Display Final Preprocessed Data

```
print("\nPreprocessed Data:")
print(df)
```

Expected Output:

```
Original Data:
      Name  Gender   Age         City
0    Alice  Female  25.0     New York
1      Bob    Male  30.0  Los Angeles
2  Charlie    Male   NaN      Chicago
3    David    Male  22.0      Chicago
4      Eva  Female  28.0        Miami



Preprocessed Data:
      Name    Age  Gender_encoded  City_Chicago  City_Los Angeles  City_Miami  City_New York
0    Alice  25.00               0             0                 0           0              1
1      Bob  30.00               1             0                 1           0              0
2  Charlie  26.25               1             1                 0           0              0
3    David  22.00               1             1                 0           0              0
4      Eva  28.00               0             0                 0           1              0
```

The missing Age was filled with the mean value (26.25), Gender was label encoded, and City was one-hot encoded for easy visualization and further analysis.

PRACTICAL NO: 6

Aim:

To apply feature scaling and normalization techniques using Python to preprocess data for improved and meaningful visualization.

Theory:

- What is Feature Scaling?

Feature scaling is the process of bringing all features (columns) to a similar scale so that no feature dominates the others due to its magnitude.

- Why is it important for visualization?
  - Ensures fair comparison across features.
  - Prevents skewed graphs due to differing units (e.g., salary vs. age).
  - Highlights patterns more clearly.

Common Techniques:

Method	Description
Min-Max Scaling	Scales values between 0 and 1 using: $(X-\min)/(\max-\min)$ (X - min) / (max - min)
Z-score Normalization (Standardization)	Centers around mean (0) with unit variance using: $(X-\mu)/\sigma$ (X - μ) / σ
MaxAbs Scaling	Scales each feature by its maximum absolute value.

These techniques are especially useful for visualization like heatmaps, PCA plots, or any multi-dimensional graph.

Code:

☑ **Step 1: Import Required Libraries**

```
import pandas as pd
import numpy as np
from sklearn.preprocessing import MinMaxScaler, StandardScaler
import matplotlib.pyplot as plt
```

☑ **Step 2: Create a Sample Dataset**

```
# Sample dataset
data = {
    'Age': [22, 25, 47, 52, 46],
    'Salary': [25000, 32000, 56000, 60000, 58000]
}

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

☑ Step 3: Apply Min-Max Scaling (Range: 0 to 1)

```
minmax_scaler = MinMaxScaler()
df_minmax = pd.DataFrame(minmax_scaler.fit_transform(df),
                         columns=['Age_Scaled', 'Salary_Scaled'])
print("\nMin-Max Scaled Data:\n", df_minmax)
```

☑ Step 4: Apply Standardization (Z-score Normalization)

```
std_scaler = StandardScaler()
df_standardized = pd.DataFrame(std_scaler.fit_transform(df),
                               columns=['Age_Std', 'Salary_Std'])
print("\nZ-score Normalized Data:\n", df_standardized)
```

☑ Step 5: Visualization — Compare Original vs Scaled Data

```
plt.figure(figsize=(10, 5))

# Original Data
plt.subplot(1, 3, 1)
plt.plot(df['Age'], label='Age')
plt.plot(df['Salary'], label='Salary')
plt.title("Original Data")
plt.legend()

# Min-Max Scaled Data
plt.subplot(1, 3, 2)
plt.plot(df_minmax['Age_Scaled'], label='Age_Scaled')
plt.plot(df_minmax['Salary_Scaled'], label='Salary_Scaled')
plt.title("Min-Max Scaled")
plt.legend()

# Standardized Data
plt.subplot(1, 3, 3)
plt.plot(df_standardized['Age_Std'], label='Age_Std')
plt.plot(df_standardized['Salary_Std'], label='Salary_Std')
plt.title("Z-score Normalized")
plt.legend()

plt.tight_layout()
plt.show()
```

Expected Output:

- Original Data: Displays raw Age and Salary values.
- Min-Max Scaled Data: Values between 0 and 1.
- Z-score Normalized Data: Mean = 0, Std Dev = 1.

Graphs:

- First plot shows original values on very different scales.
- Second (Min-Max) and third (Z-score) plots show features on similar scales, making it easier to compare trends visually.

PRACTICAL NO: 7

Aim:

To create bar charts, line graphs, and scatter plots using Matplotlib and Seaborn libraries in Python for effective data visualization.

Theory:

Visualization is an essential part of data analysis that helps in understanding trends, relationships, and patterns within data. Python offers libraries like Matplotlib and Seaborn to create high-quality plots.

- Matplotlib:
  - A low-level, highly customizable plotting library.
  - Suitable for static, basic plots.
- Seaborn:
  - Built on top of Matplotlib.
  - Provides a high-level interface for statistical graphics.
  - Prettier and easier for certain types of plots, especially with pandas DataFrames.
- Types of Plots Covered:

Plot Type	Purpose
Bar Chart	Show comparisons between categories.
Line Graph	Show trends over a continuous variable like time.
Scatter Plot	Show relationship between two numerical variables.

Code:

☑ Step 1: Import Required Libraries

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

☑ Step 2: Create a Sample Dataset

```
# Sample data
data = {
    'Month': ['Jan', 'Feb', 'Mar', 'Apr', 'May'],
    'Sales': [200, 250, 300, 280, 320],
    'Profit': [20, 35, 50, 40, 60]
}

df = pd.DataFrame(data)
```

☑ Step 3: Bar Chart (Matplotlib)

```
plt.figure(figsize=(6, 4))
plt.bar(df['Month'], df['Sales'], color='skyblue')
plt.title("Monthly Sales (Bar Chart - Matplotlib)")
plt.xlabel("Month")
plt.ylabel("Sales")
plt.grid(True, linestyle='--', alpha=0.5)
plt.show()
```

☑ Step 4: Line Graph (Seaborn)

```
plt.figure(figsize=(6, 4))
sns.lineplot(x='Month', y='Sales', data=df, marker='o', label='Sales')
sns.lineplot(x='Month', y='Profit', data=df, marker='s', label='Profit')
plt.title("Sales and Profit Over Months (Line Graph - Seaborn)")
plt.ylabel("Value")
plt.legend()
plt.grid(True, linestyle='--', alpha=0.5)
plt.show()
```

☑ Step 5: Scatter Plot (Matplotlib)

```
plt.figure(figsize=(6, 4))
plt.scatter(df['Sales'], df['Profit'], color='red')
plt.title("Sales vs Profit (Scatter Plot - Matplotlib)")
plt.xlabel("Sales")
plt.ylabel("Profit")
plt.grid(True, linestyle='--', alpha=0.5)
plt.show()
```

Expected Output:

1. Bar Chart (Matplotlib) – Displays sales values per month as vertical bars.
2. Line Graph (Seaborn) – Two lines showing trends of sales and profit across months.
3. Scatter Plot (Matplotlib) – Shows the relationship between sales and profit; each point is one month.

These visualizations help:

- Compare monthly performance.
- See trends in growth or decline.
- Observe correlation between sales and profit.

PRACTICAL NO: 8

Aim:

To visualize the **distribution of data** using **histograms**, **box plots**, and **violin plots** with the **Seaborn** library in Python.

Theory:

Understanding the distribution of data is essential for statistical analysis and decision-making. Seaborn provides multiple tools for this purpose:

Plot Type	Description
Histogram	Shows the frequency of data within intervals. Helps understand the shape of the distribution.
Box Plot	Summarizes data with median, quartiles, and outliers using a simple box-and-whisker diagram.
Violin Plot	Combines box plot with a kernel density estimate to show the distribution's shape.

Code:

☑ Step 1: Import Required Libraries

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

☑ Step 2: Load Example Dataset

```
# Load Seaborn's built-in 'tips' dataset
tips = sns.load_dataset("tips")
```

☑ Step 3: Histogram of Total Bill

```
plt.figure(figsize=(6, 4))
sns.histplot(data=tips, x="total_bill", kde=True, color="skyblue")
plt.title("Histogram of Total Bill")
plt.xlabel("Total Bill")
plt.ylabel("Frequency")
plt.grid(True, linestyle='--', alpha=0.5)
plt.show()
```

☑ Step 4: Box Plot of Total Bill by Day

```
plt.figure(figsize=(6, 4))
sns.boxplot(data=tips, x="day", y="total_bill", palette="Set2")
```

```
plt.title("Box Plot of Total Bill by Day")
plt.xlabel("Day")
plt.ylabel("Total Bill")
plt.grid(True, linestyle='--', alpha=0.5)
plt.show()
```

☑ Step 5: Violin Plot of Total Bill by Gender

```
plt.figure(figsize=(6, 4))
sns.violinplot(data=tips, x="sex", y="total_bill", palette="muted")
plt.title("Violin Plot of Total Bill by Gender")
plt.xlabel("Gender")
plt.ylabel("Total Bill")
plt.grid(True, linestyle='--', alpha=0.5)
plt.show()
```

Expected Output:

1. **Histogram** – Displays the frequency distribution of the total_bill column with a smooth KDE line overlay.

2. **Box Plot** – Shows median, quartiles, and outliers of total_bill grouped by day (Thu, Fri, Sat, Sun).

3. **Violin Plot** – Visualizes both the box plot and the distribution (KDE) of total_bill grouped by sex.

These plots help analyze central tendency, spread, and variation between groups in a dataset.

PRACTICAL NO: 9

Aim:

To create advanced data visualizations such as heatmaps and pair plots using the Seaborn library to explore and analyze relationships between multiple variables in a dataset.

Theory:

Visualizing relationships between multiple features in a dataset helps in identifying correlations, clusters, and outliers. Seaborn offers two powerful tools for this:

- Heatmap:
  - A color-coded matrix that displays the correlation coefficients between numerical variables.
  - Values close to +1 or -1 indicate strong relationships.
- Pair Plot:
  - Shows pairwise relationships between all numeric variables in the dataset using scatter plots and histograms.
  - Useful for detecting patterns, correlations, and clusters.

Use Cases:

- Identify strongly correlated variables.
- Explore how variables relate to each other.
- Detect multicollinearity before applying machine learning models.

Code:

☑ Step 1: Import Required Libraries

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

☑ Step 2: Load Example Dataset

```
# Load Seaborn's built-in 'iris' dataset
iris = sns.load_dataset("iris")
```

☑ Step 3: Heatmap of Correlation Matrix

```
plt.figure(figsize=(8, 6))
correlation_matrix = iris.corr(numeric_only=True)
sns.heatmap(correlation_matrix, annot=True, cmap="coolwarm", linewidths=0.5)
plt.title("Heatmap - Correlation Matrix (Iris Dataset)")
plt.show()
```

☑ Step 4: Pair Plot

```
sns.pairplot(iris, hue="species", palette="Set1")
plt.suptitle("Pair Plot - Iris Dataset", y=1.02)
plt.show()
```

Expected Output:

1. **Heatmap** – A grid showing correlation values between features like sepal_length, petal_length, etc. The color intensity indicates the strength and direction of the relationship.

2. **Pair Plot –** A matrix of scatter plots showing how each feature correlates with others, with color coding by species (setosa, versicolor, virginica).

These plots help you:

- Detect linear relationships (e.g., petal_length vs petal_width).

- Visually separate classes (e.g., how species differ based on measurements).

PRACTICAL NO: 10

Aim:

To design and implement an interactive dashboard using Plotly and Dash in Python for dynamic data visualization and real-time interaction.

Theory:

- What is Dash?
  - Dash is a Python framework built by Plotly for building web-based interactive dashboards.
  - Combines Flask (for backend), Plotly.js (for visualizations), and React.js (for front-end interactivity).
  - Allows data scientists to build interactive web apps with pure Python.

- What is Plotly?
  - Plotly is a graphing library that enables the creation of interactive and web-ready visualizations like line charts, bar graphs, scatter plots, etc.

Advantages of Dash:

- No need to learn HTML, CSS, or JavaScript.
- Fully interactive and responsive dashboards.
- Good for displaying real-time data or user-driven insights.

Code:

☑ **Step 1: Install Required Libraries**

Run this in your terminal or command prompt before executing the app:

pip install dash plotly seaborn pandas

☑ **Step 2: Full Python Code (app.py)**

```
import dash
from dash import dcc, html
from dash.dependencies import Input, Output
import plotly.express as px
import seaborn as sns
import pandas as pd

# ----------------------------
# Load dataset
# ----------------------------
df = sns.load_dataset('iris')

# ----------------------------
# Initialize Dash app
# ----------------------------
```

```
app = dash.Dash(__name__)

# ----------------------------
# Layout of the dashboard
# ----------------------------
app.layout = html.Div([
    html.H1(" ✿ Iris Species Dashboard", style={'textAlign': 'center', 'color': '#2E86C1'}),

    html.Label("Select Species:", style={'fontWeight': 'bold'}),

    dcc.Dropdown(
        id='species-dropdown',
        options=[{'label': species.capitalize(), 'value': species} for species in df['species'].unique()],
        value='setosa',
        clearable=False,
        style={'width': '50%', 'marginBottom': '20px'}
    ),

    dcc.Graph(id='scatter-plot')
])

# ----------------------------
# Callback to update graph
# ----------------------------
@app.callback(
    Output('scatter-plot', 'figure'),
    Input('species-dropdown', 'value')
)
def update_graph(selected_species):
    filtered_df = df[df['species'] == selected_species]
    fig = px.scatter(
        filtered_df,
        x='sepal_length',
        y='sepal_width',
        color='species',
        title=f"Sepal Length vs Sepal Width for {selected_species.capitalize()}",
        labels={'sepal_length': 'Sepal Length', 'sepal_width': 'Sepal Width'},
        template='plotly_dark'
    )
    return fig
```

```
# ----------------------------
# Run the app
# ----------------------------
if __name__ == '__main__':
    app.run_server(debug=True)
```

Expected Output:

- A web-based dashboard hosted at http://127.0.0.1:8050/
- Dropdown menu to select a species (setosa, versicolor, virginica)
- A live interactive scatter plot that updates based on the selected species
- Hovering over points shows exact values
- Users can zoom, pan, and export the plot as an image