AI & ML FOUNDATION COURSE

www.voltusacademy.in

VOLTUS AI ACADEMY

Matplotlib

Matplotlib is a popular Python library used for creating static, animated, and interactive visualizations.

It provides a wide variety of plotting options and can be used in conjunction with NumPy and pandas for data manipulation and analysis.

How to use Matplotlib

```
import matplotlib as plt
```

Matplotlib Pyplot

Most of the Matplotlib utilities lies under the pyplot submodule, and are usually imported under the plt alias:

```
import matplotlib.pyplot as plt
```

Matplotlib supports a wide variety of plot types to visualize data effectively. Here are some of the most commonly used types of plots available in Matplotlib:

1. **Line Plot**: Used to visualize the relationship between two variables by connecting data points with straight lines.

2. **Scatter Plot**: Displays the relationship between two variables as a set of points, where each point represents an observation in the data.

3. **Bar Plot**: Represents categorical data with rectangular bars where the height or length of the bar corresponds to the value it represents.

4. **Histogram**: Shows the distribution of a dataset by grouping data into bins of equal width and plotting the frequency of observations in each bin.

5. **Pie Chart**: Represents data as slices of a circular pie, where the size of each slice corresponds to the proportion of data it represents.

6. **Box Plot (Box-and-Whisker Plot)**: Summarizes the distribution of a dataset by showing the median, quartiles, and outliers.

7. **Violin Plot**: Combines aspects of box plot and density plot, showing the distribution of the data across different levels of a categorical variable.

8. **Heatmap**: Visualizes data in a matrix where each value is represented by a color. Useful for showing correlations or distributions across two categorical variables.

1. **Line Plot:**

   - **Representation**: A line plot connects data points with straight lines.
   - **Usage**: Typically used to show how data changes over continuous intervals or time.
   - **Data Type**: Best suited for continuous data where the order of points matters (e.g., time series data).
   - **Interpretation**: Useful for visualizing trends, patterns, or relationships between variables that have a sequential or ordered relationship.
   - **Example**: Plotting stock prices over time, temperature variations throughout the day, or population growth over years.

Plotting x and y points

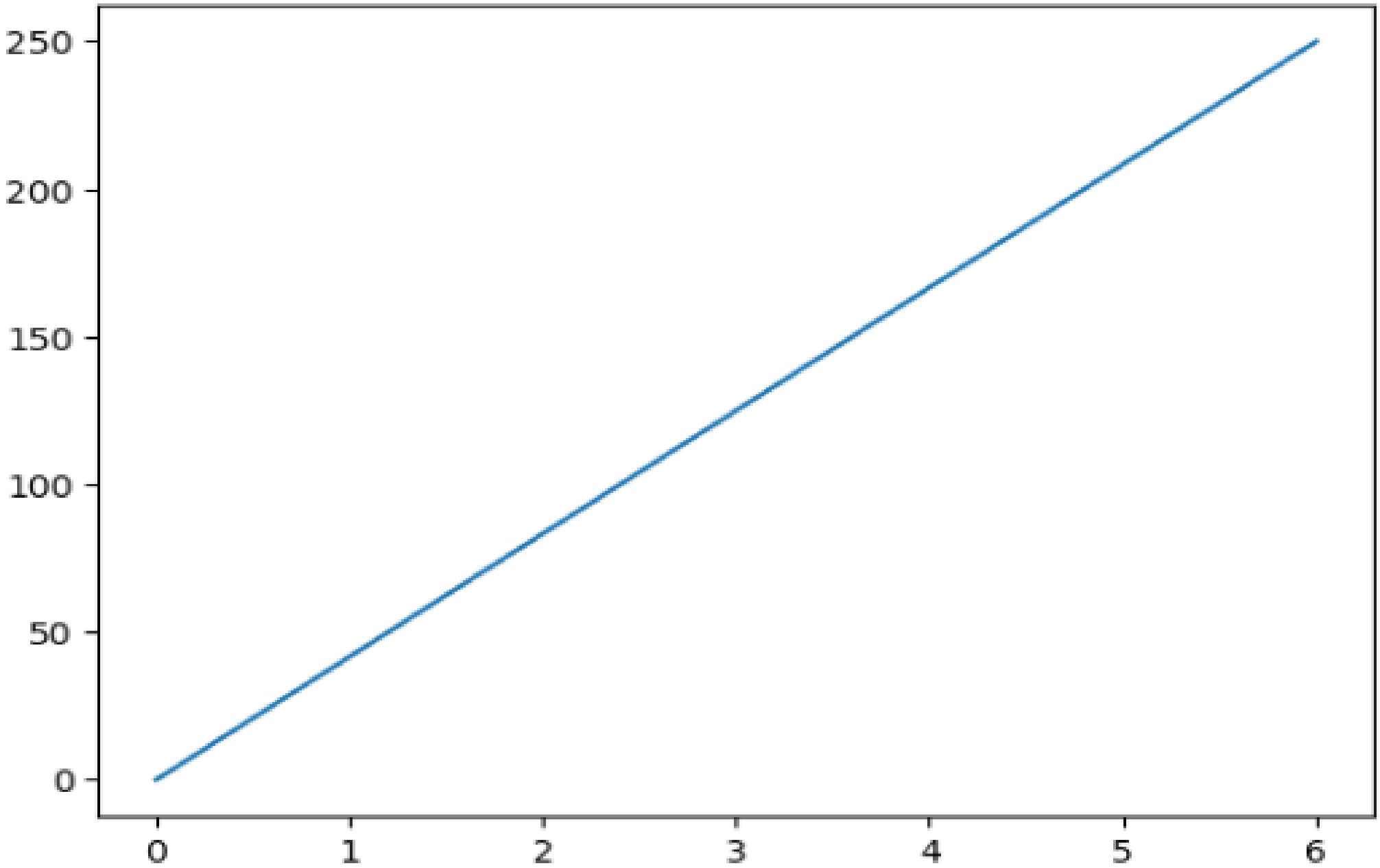
- The plot() function is used to draw points (markers) in a diagram.
- By default, the **plot() function draws a line from point to point**.
- The function takes parameters for specifying points in the diagram.
- Parameter 1 is an array containing the points on the x-axis (horizontal axis).
- Parameter 2 is an array containing the points on the y-axis (vertical axis)..

Plotting Line

Draw a line in a diagram from position (0,0) to position (6,250):

[1]:

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

x=np.array([0,6])
y=np.array([0,250])

plt.plot(x,y)
plt.show()
```


250
200
150
100
50
0
0
1
2
3
4
5
6

1. **Imports:**

   - `` `import matplotlib.pyplot as plt` ``: Imports the pyplot module from Matplotlib, which is used for plotting.
   - `` `import numpy as np` ``: Imports the NumPy library, which is used here to create arrays.

2. **Data Creation:**

   - `` `x = np.array([0, 6])` ``: Defines the x-coordinates of the line. Starts at 0 and ends at 6.
   - `` `y = np.array([0, 250])` ``: Defines the y-coordinates of the line. Starts at 0 and ends at 250.

3. **Plotting:**

   - `` `plt.plot(x, y)` ``: Plots the line using the x and y coordinates defined earlier.
   - By default, `` `plt.plot()` `` connects the points (0, 0) and (6, 250) with a straight line.

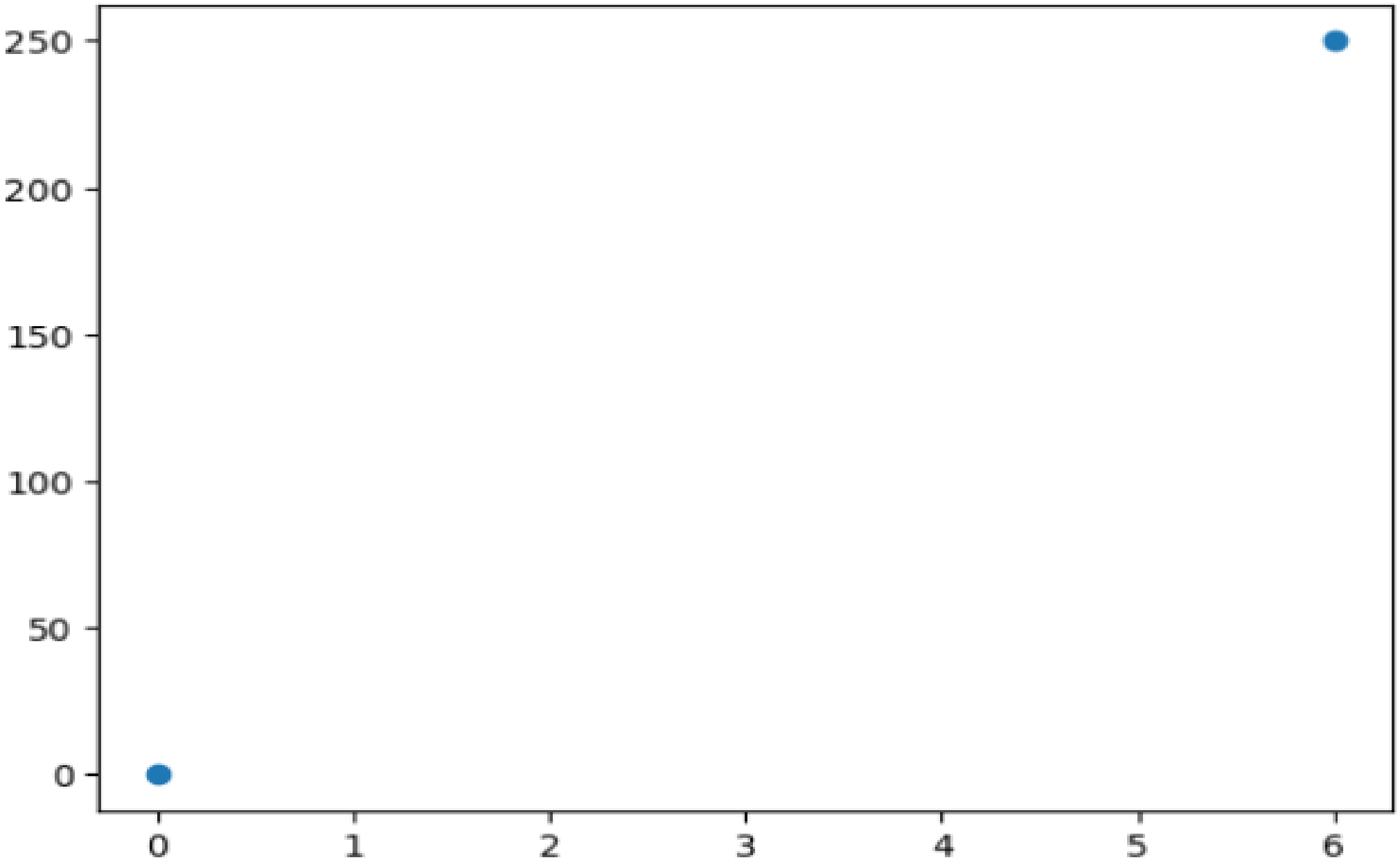
4. Displaying the Plot:

- `plt.show()`: Displays the plot on the screen.

Plotting without Line

To plot only the markers, you can use shortcut string notation parameter 'o', which means 'rings'.

[51]:
```
import matplotlib.pyplot as plt
import numpy as np

x=np.array([0,6])
y=np.array([0,250])

plt.plot(x,y,"o")
plt.show()
```


250
200
150
100
50
0
0
1
2
3
4
5
6

1. Imports:

   - `import matplotlib.pyplot as plt`: Imports the pyplot module from Matplotlib for plotting.
   - `import numpy as np`: Imports the NumPy library for array operations.

2. Data Creation:

   - `x = np.array([0, 6])`: Defines the x-coordinates of the points. Starts at 0 and ends at 6.
   - `y = np.array([0, 250])`: Defines the y-coordinates of the points. Starts at 0 and ends at 250.

3. **Plotting**:

- `plt.plot(x, y, "o")`: Plots the points defined by `x` and `y`.
  - The third argument `"o"` specifies that markers (circles) should be used at each point `(x[i], y[i])`.
  - This changes the plot from a connected line to discrete markers.

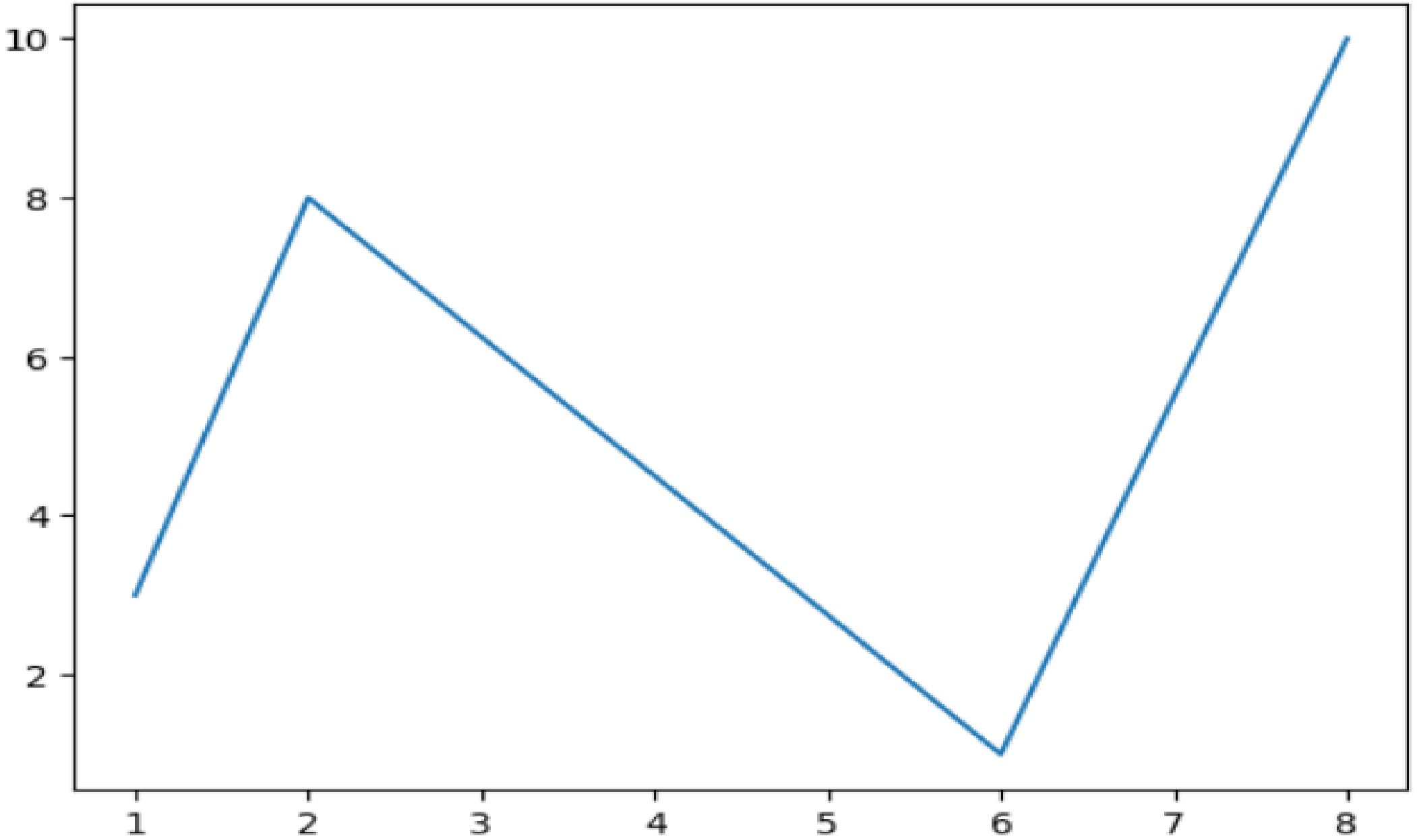
4. **Displaying the Plot**:

- `plt.show()`: Displays the plot on the screen.

Multiple Point line Plot

[2]:

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

x=np.array([1,2,6,8])
y=np.array([3,8,1,10])

plt.plot(x,y)
plt.show()
```


10
8
6
4
2
1
2
3
4
5
6
7
8

1. **Imports**:

   - `import matplotlib.pyplot as plt`: Imports the pyplot module from Matplotlib for plotting.
   - `import numpy as np`: Imports the NumPy library for array operations.

2. **Data Creation**:

   - `x = np.array([1, 2, 6, 8])`: Defines the x-coordinates of the data points `[1, 2, 6, 8]`.
   - `y = np.array([3, 8, 1, 10])`: Defines the y-coordinates of the data points `[3, 8, 1, 10]`.

3. Plotting:

- `plt.plot(x, y)`: Plots the points defined by `x` and `y`.
    - By default, `plt.plot()` connects the points with straight line segments in the order they are given in the arrays `x` and `y`.

4. Displaying the Plot:

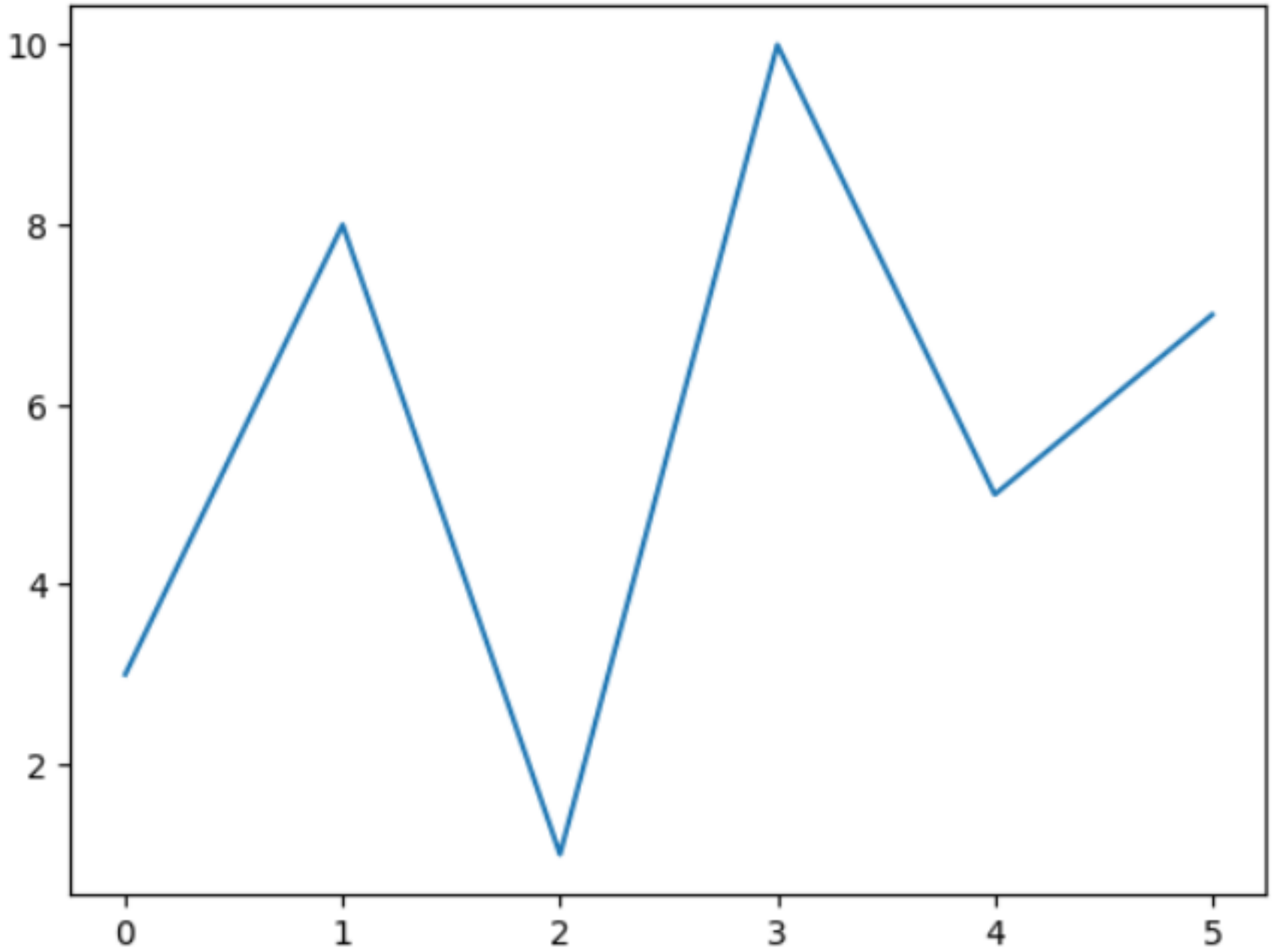
- `plt.show()`: Displays the plot on the screen.

Default X-Axis

If we do not specify the points in the x-axis, they will get the default values 0, 1, 2, 3, (etc. depending on the length of the y-points.

[3]:

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

y=np.array([3,8,1,10,5,7])

plt.plot(y)
plt.show()
```


10
8
6
4
2
0
1
2
3
4
5

1. Imports:

   - `import matplotlib.pyplot as plt`: Imports the pyplot module from Matplotlib for plotting.
   - `import numpy as np`: Imports the NumPy library for array operations.

2. Data Creation:

   - `y = np.array([3, 8, 1, 10, 5, 7])`: Defines an array `y` with six data points `[3, 8, 1, 10, 5, 7]`.

3. **Plotting**:

   - `plt.plot(y)`: Plots the values in the array `y`.
     - When only one array (`y`) is provided to `plt.plot()`, it assumes these values represent the y-coordinates of points plotted against their index positions as the x-coordinates (`[0, 1, 2, ..., N-1]` where `N` is the number of elements in `y`).
     - Therefore, the plot will connect these points with straight line segments.

4. **Displaying the Plot**:

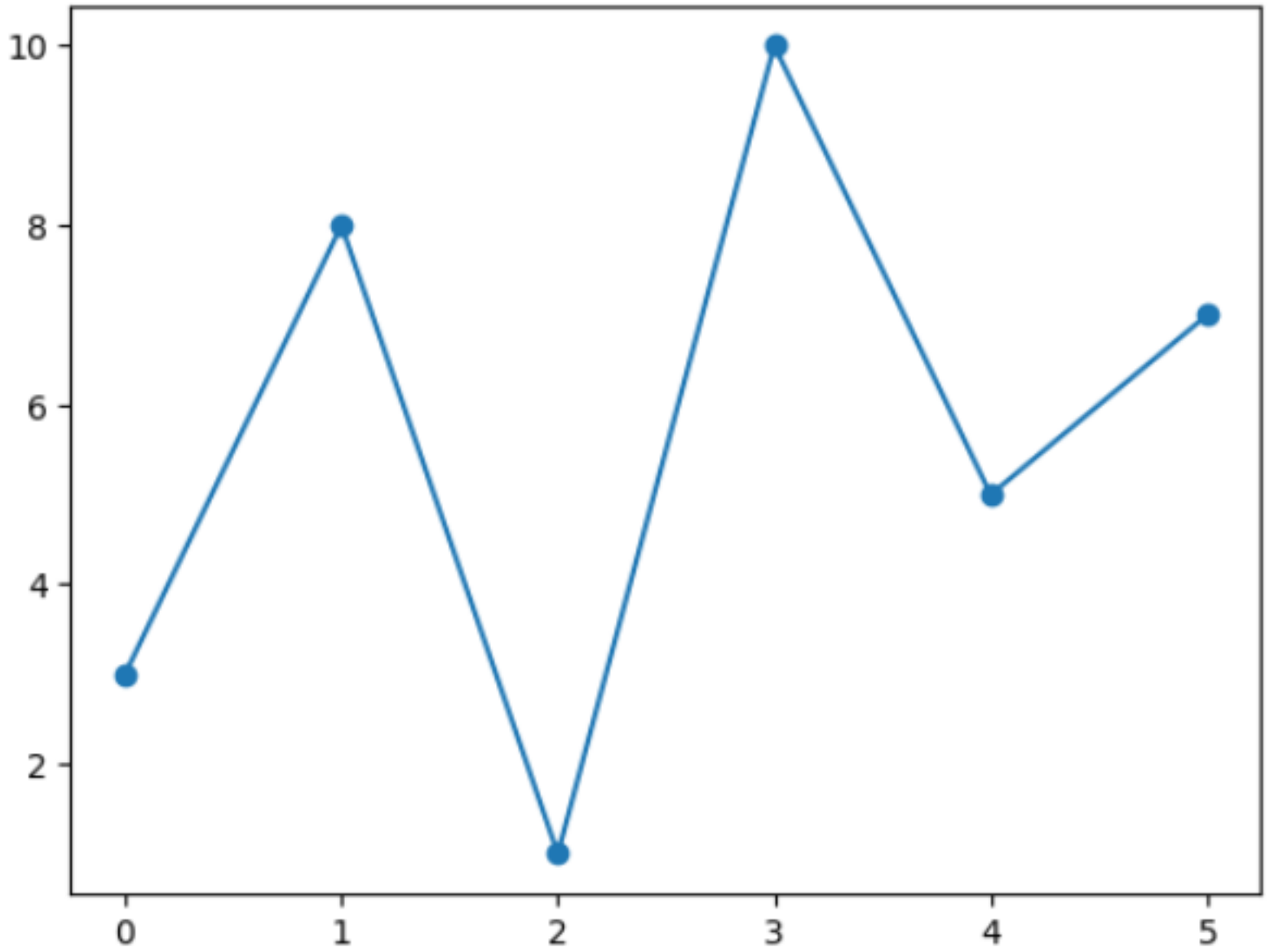
   - `plt.show()`: Displays the plot on the screen.

Matplotlib Markers

You can use the keyword argument marker to emphasize each point with a specified marker:

[4]:

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

y=np.array([3,8,1,10,5,7])

plt.plot(y,marker='o')
plt.show()
```


10
8
6
4
2
0
1
2
3
4
5

1. Imports:

   - `import matplotlib.pyplot as plt`: Imports the pyplot module from Matplotlib for plotting.
   - `import numpy as np`: Imports the NumPy library for array operations.

2. Data Creation:

   - `y = np.array([3, 8, 1, 10, 5, 7])`: Defines an array `y` with six data points `[3, 8, 1, 10, 5, 7]`.

3. **Plotting:**

   - `plt.plot(y, marker='o')`: Plots the values in the array `y` with circular markers (`'o'`) at each data point.
     - The `marker='o'` argument specifies that circular markers should be used.
     - By default, `plt.plot()` connects the points with straight lines, and now each point is marked with a circle.

4. **Displaying the Plot:**

   - `plt.show()`: Displays the plot on the screen.

Markers Reference

Marker	Description
'o'	Circle
'*'	Star
'.'	Point
'x'	X
'X'	X (filled)
'+'	Plus
'P'	Plus (filled)
's'	Square
'D'	Diamond

Marker	Description
'd'	Diamond (thin)
'p'	Pentagon
'H'	Hexagon
'h'	Hexagon
'v'	Triangle Down
'^'	Triangle Up
'<'	Triangle Left
'>'	Triangle Right

Line Reference

Line Syntax	Description
'-'	Solid Line
':'	Dotted Line
'--'	Dashed Line
'-.'	Dashed/ Dotted Line

Color Reference

Color Syntax	Description
‘r'	Red
‘g'	Green
‘b'	Blue
‘c'	Cyan
‘m’	Magenta
‘y’	Yellow
‘k’	Black
‘w’	White

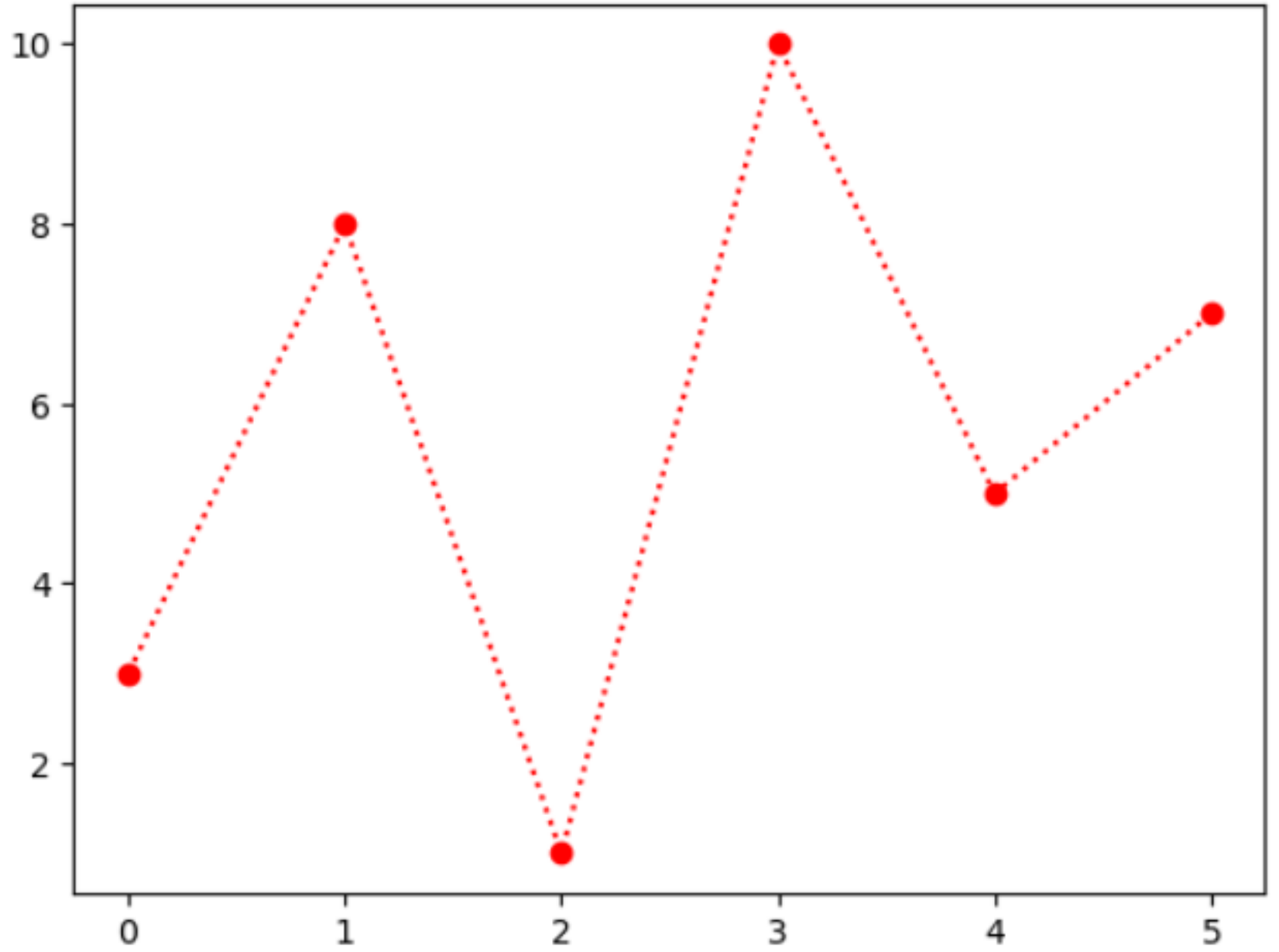
Format Strings fmt

This parameter is also called fmt, and is written with this syntax:

marker|line|color

[5]:

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

y=np.array([3,8,1,10,5,7])

plt.plot(y,"o:r")
plt.show()
```


10
8
6
4
2
0
1
2
3
4
5

1. Imports:

   - `import matplotlib.pyplot as plt`: Imports the pyplot module from Matplotlib for plotting.
   - `import numpy as np`: Imports the NumPy library for array operations.

2. Data Creation:

   - `y = np.array([3, 8, 1, 10, 5, 7])`: Defines an array `y` with six data points `[3, 8, 1, 10, 5, 7]`.

3. **Plotting:**

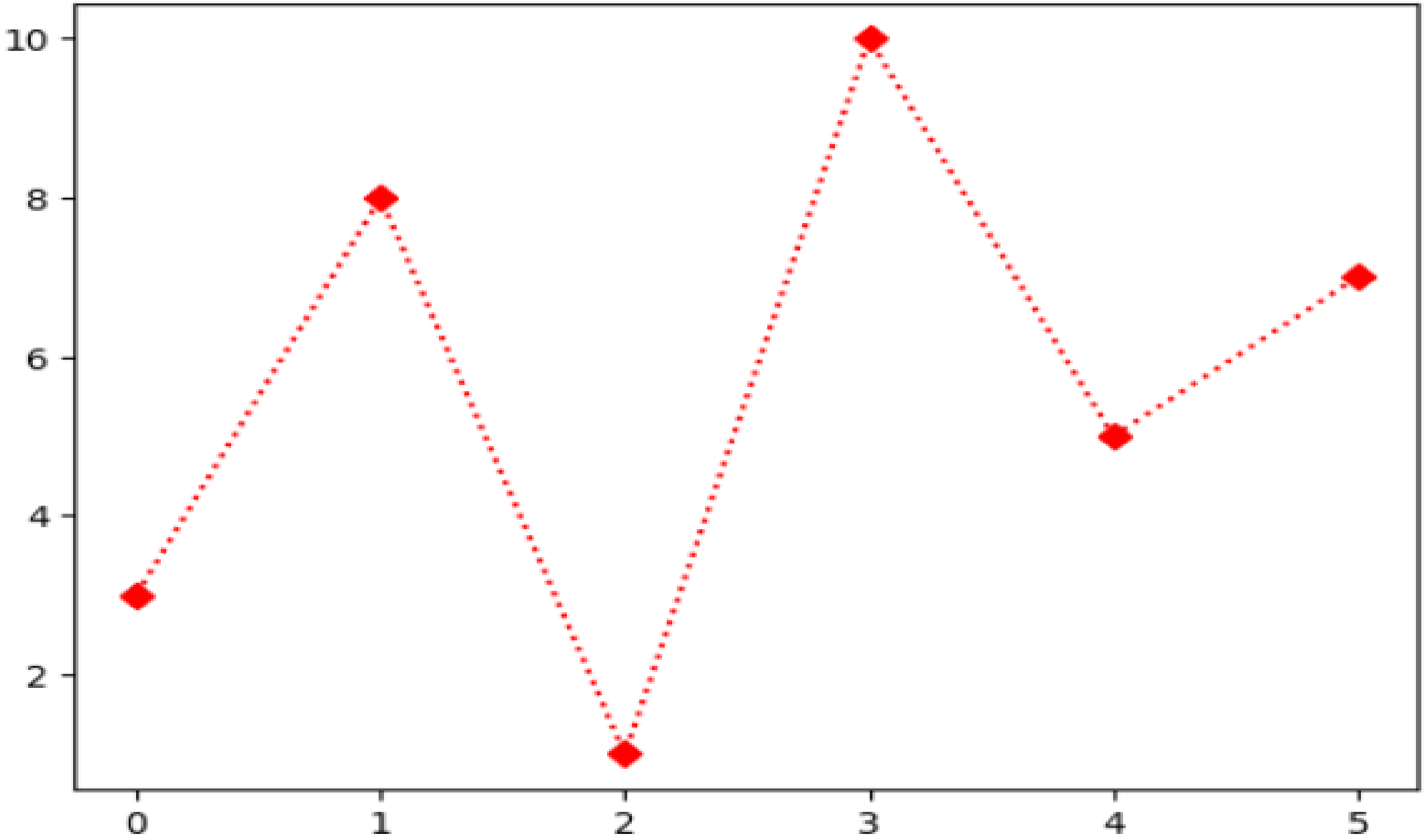
   - `plt.plot(y, "o:r")`: Plots the values in the array `y`.
     - The format string `"o:r"` specifies:
       - `"o"`: Marker style as a circle (`'o'`).
       - `":"`: Line style as dotted (`':'`).
       - `"r"`: Color of the line and markers as red (`'r'`).

4. **Displaying the Plot:**

   - `plt.show()`: Displays the plot on the screen.

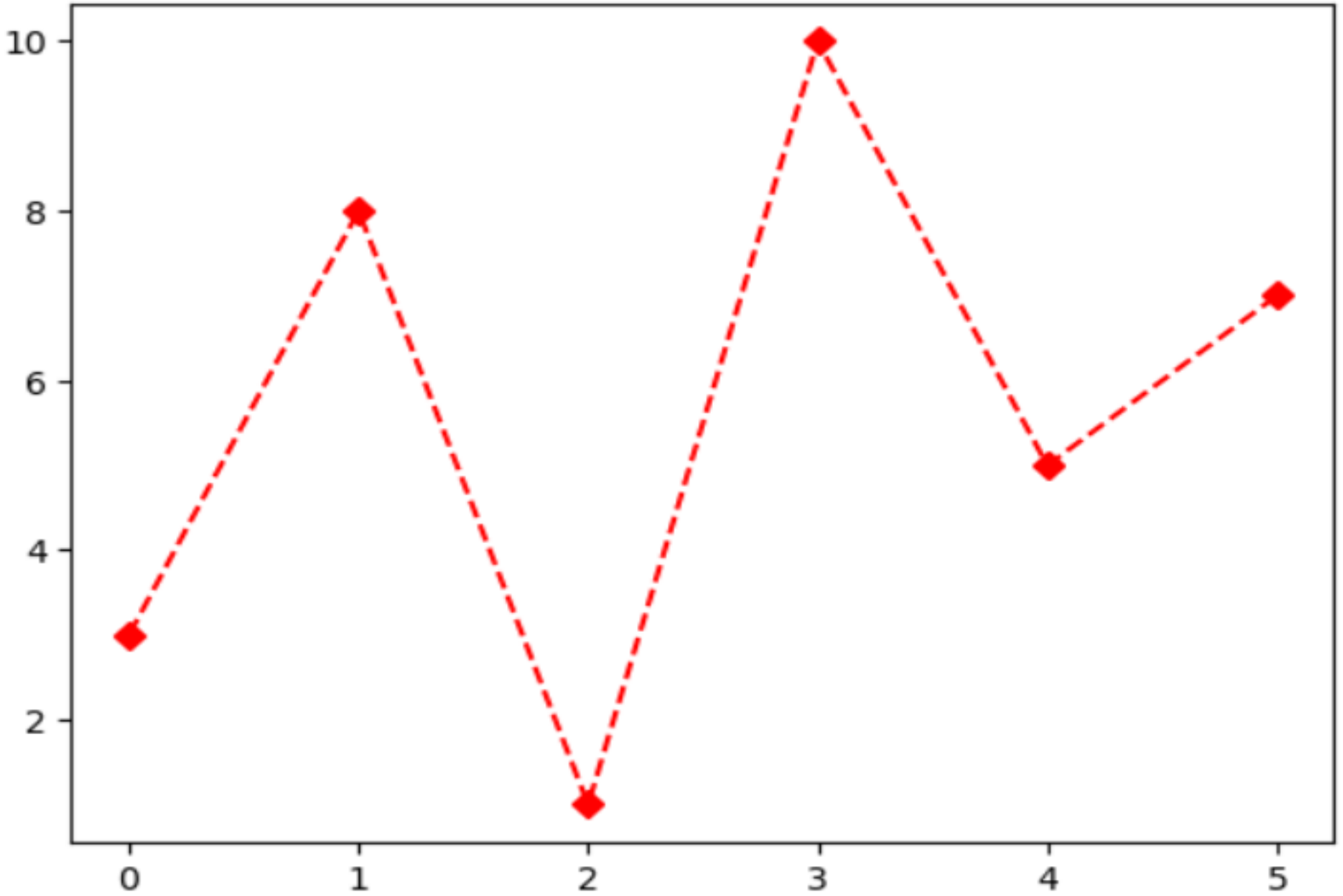
[1]:

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

y=np.array([3,8,1,10,5,7])

plt.plot(y,"D:r")
plt.show()
```


10
8
6
4
2
0
1
2
3
4
5

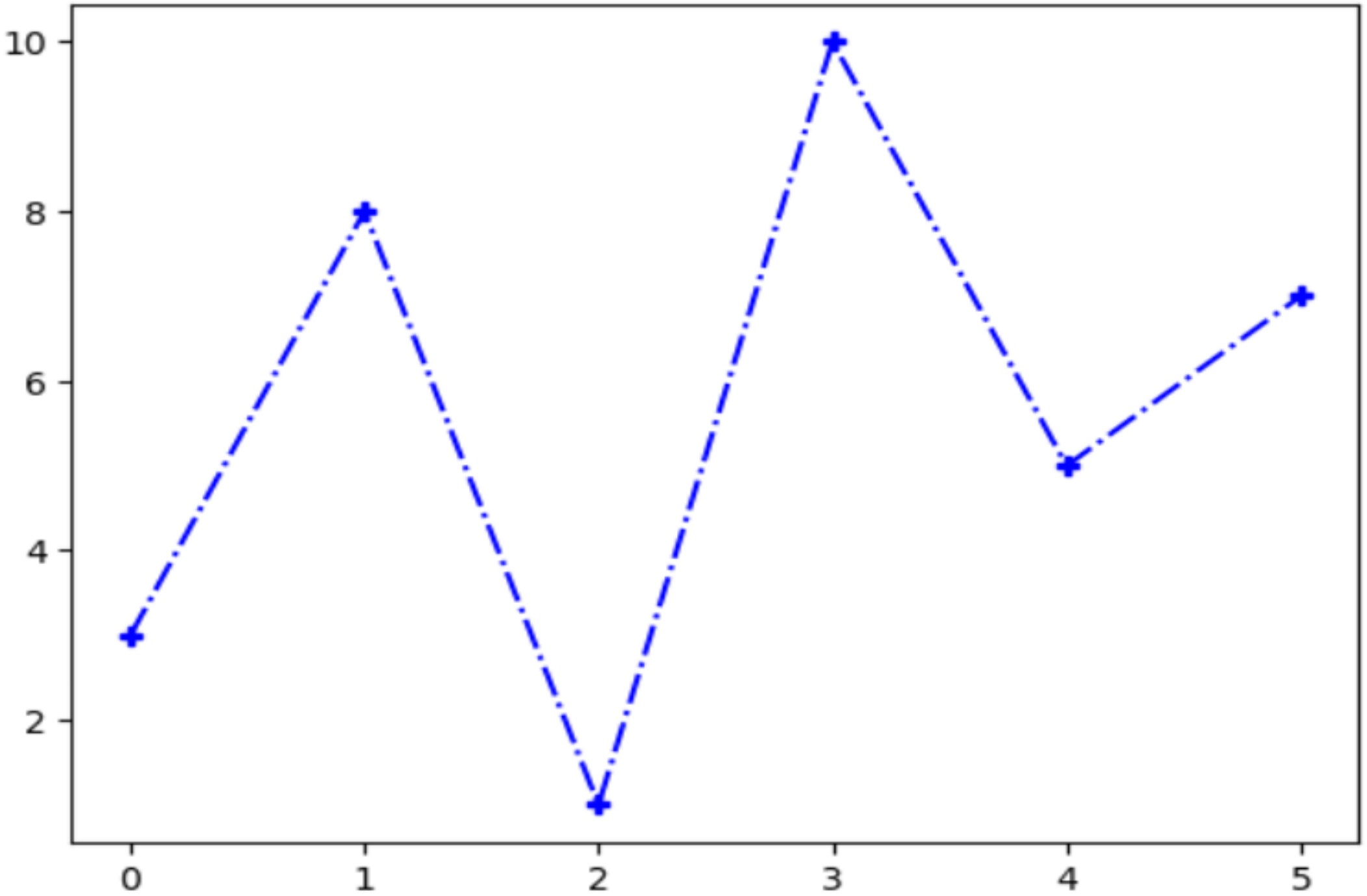
[3]:

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

y=np.array([3,8,1,10,5,7])

plt.plot(y,"D--r")
plt.show()
```


10
8
6
4
2
0
1
2
3
4
5

[1]:

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

y=np.array([3,8,1,10,5,7])

plt.plot(y,"P-.b")
plt.show()
```

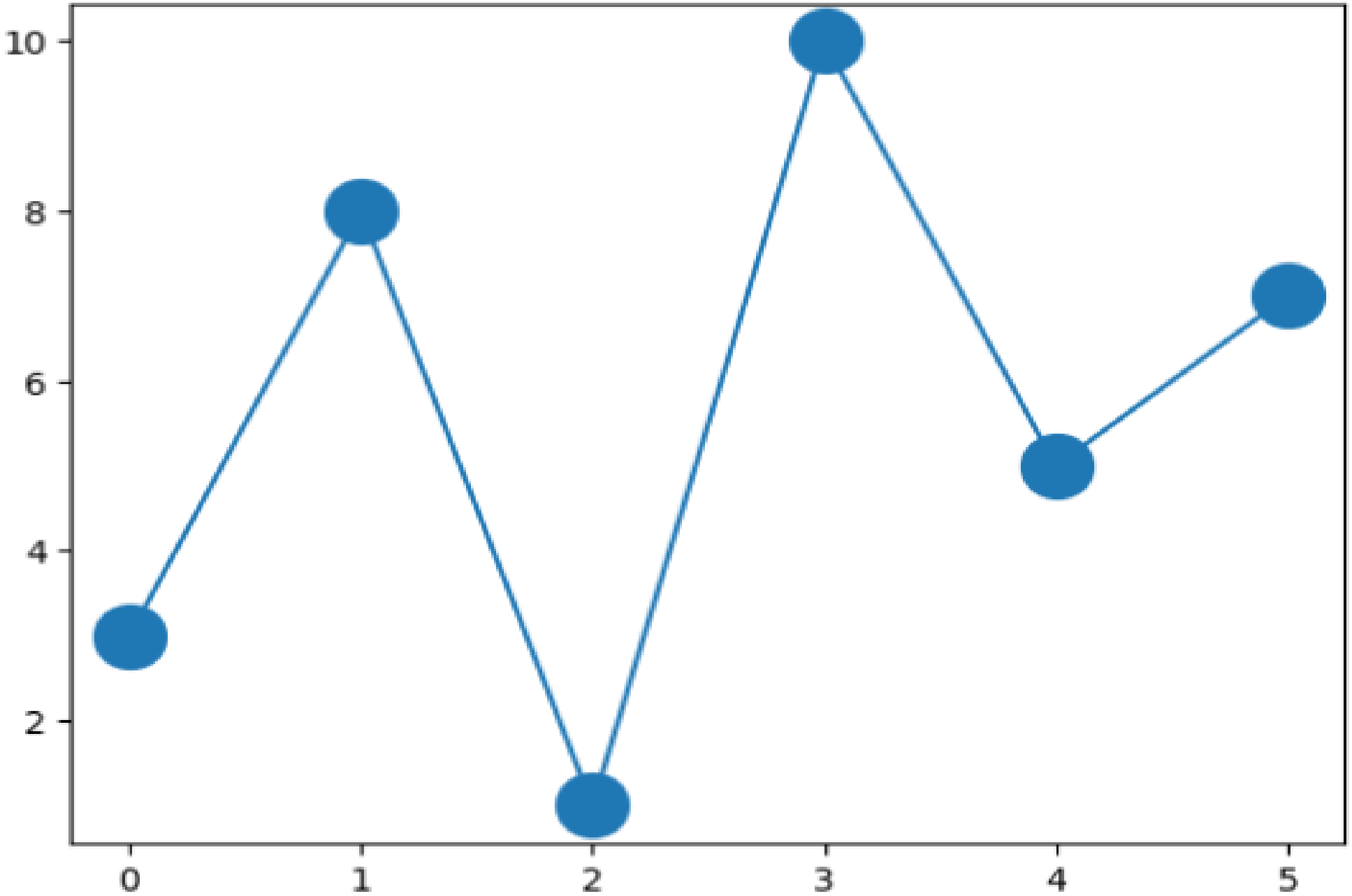

10
8
6
4
2
0
1
2
3
4
5

Markers Size

You can use the keyword argument marker size or the shorter version, *ms* to set the size of the markers:

[52]:

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

y=np.array([3,8,1,10,5,7])

plt.plot(y,marker='o',ms=20)
plt.show()
```


10
8
6
4
2
0
1
2
3
4
5

1. Imports:

   - `import matplotlib.pyplot as plt`: Imports the pyplot module from Matplotlib for plotting.
   - `import numpy as np`: Imports the NumPy library for array operations.

2. Data Creation:

   - `y = np.array([3, 8, 1, 10, 5, 7])`: Defines an array `y` with six data points `[3, 8, 1, 10, 5, 7]`.

3. **Plotting:**

- `plt.plot(y, marker='o', ms=20)`: Plots the values in the array `y`.
  - `marker='o'`: Specifies circular markers (`'o'`) at each data point.
  - `ms=20`: Sets the marker size (`ms`) to 20 points.
    - Marker size is specified in points squared (area), so `ms=20` makes the markers relatively large.

4. **Displaying the Plot:**

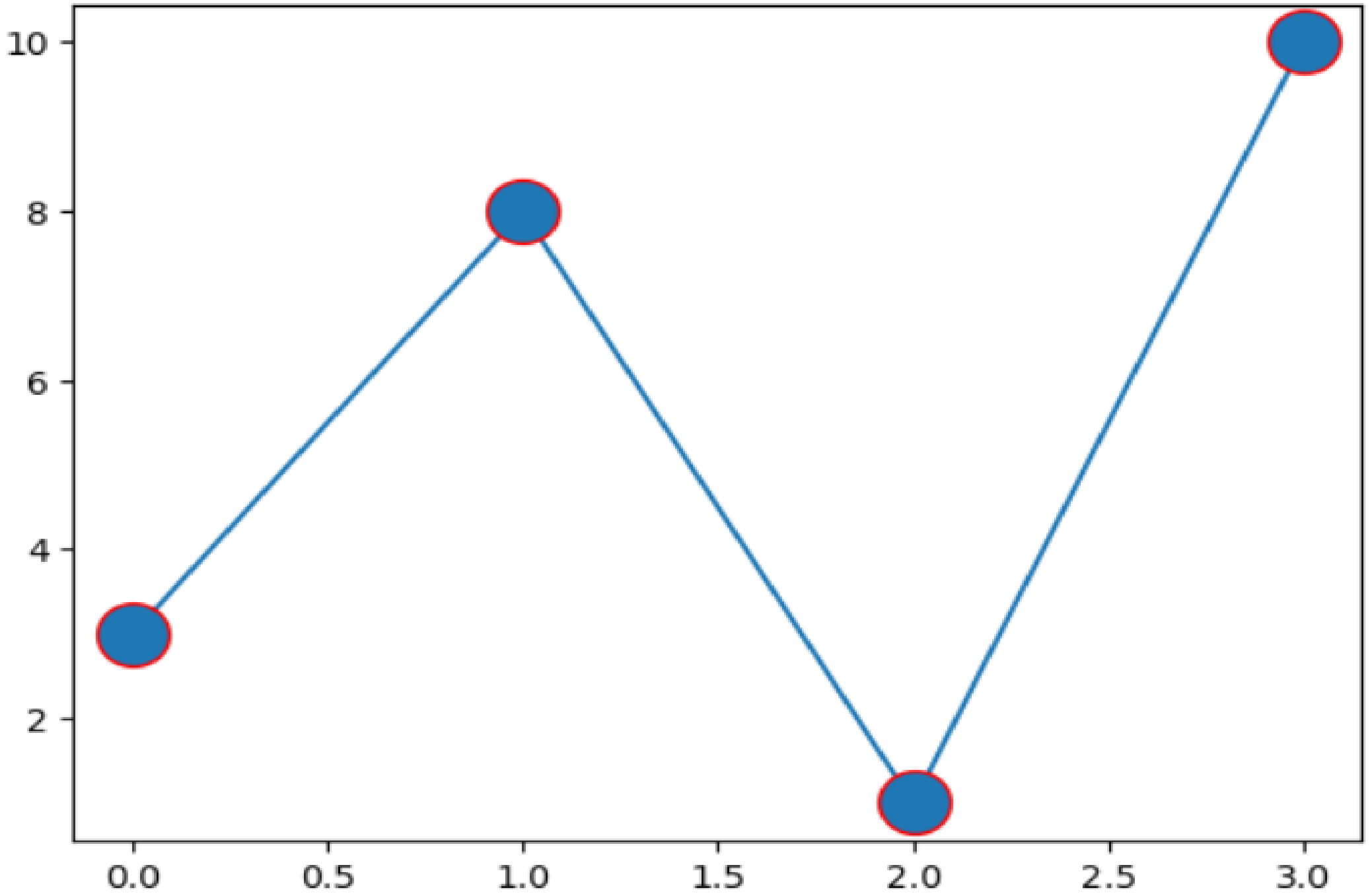
- `plt.show()`: Displays the plot on the screen.

Markers Edge Size

You can use the keyword argument markeredgecolor or the shorter mec to set the color of the edge of the markers:

[7]:

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

y=np.array([3,8,1,10])

plt.plot(y,marker='o',ms=20,mec="r")
plt.show()
```


10
8
6
4
2
0.0
0.5
1.0
1.5
2.0
2.5
3.0

1. Imports:

   - `import matplotlib.pyplot as plt`: Imports the pyplot module from Matplotlib for plotting.
   - `import numpy as np`: Imports the NumPy library for array operations.

2. Data Creation:

   - `y = np.array([3, 8, 1, 10])`: Defines an array `y` with four data points `[3, 8, 1, 10]`.

3. **Plotting:**

- `plt.plot(y, marker='o', ms=20, mec="r")`: Plots the values in the array `y`.
  - `marker='o'`: Specifies circular markers (`'o'`) at each data point.
  - `ms=20`: Sets the marker size (`ms`) to 20 points.
  - `mec="r"`: Sets the marker edge color (`mec`) to red (`'r'`).
    - This means the markers will have red edges.

4. **Displaying the Plot:**

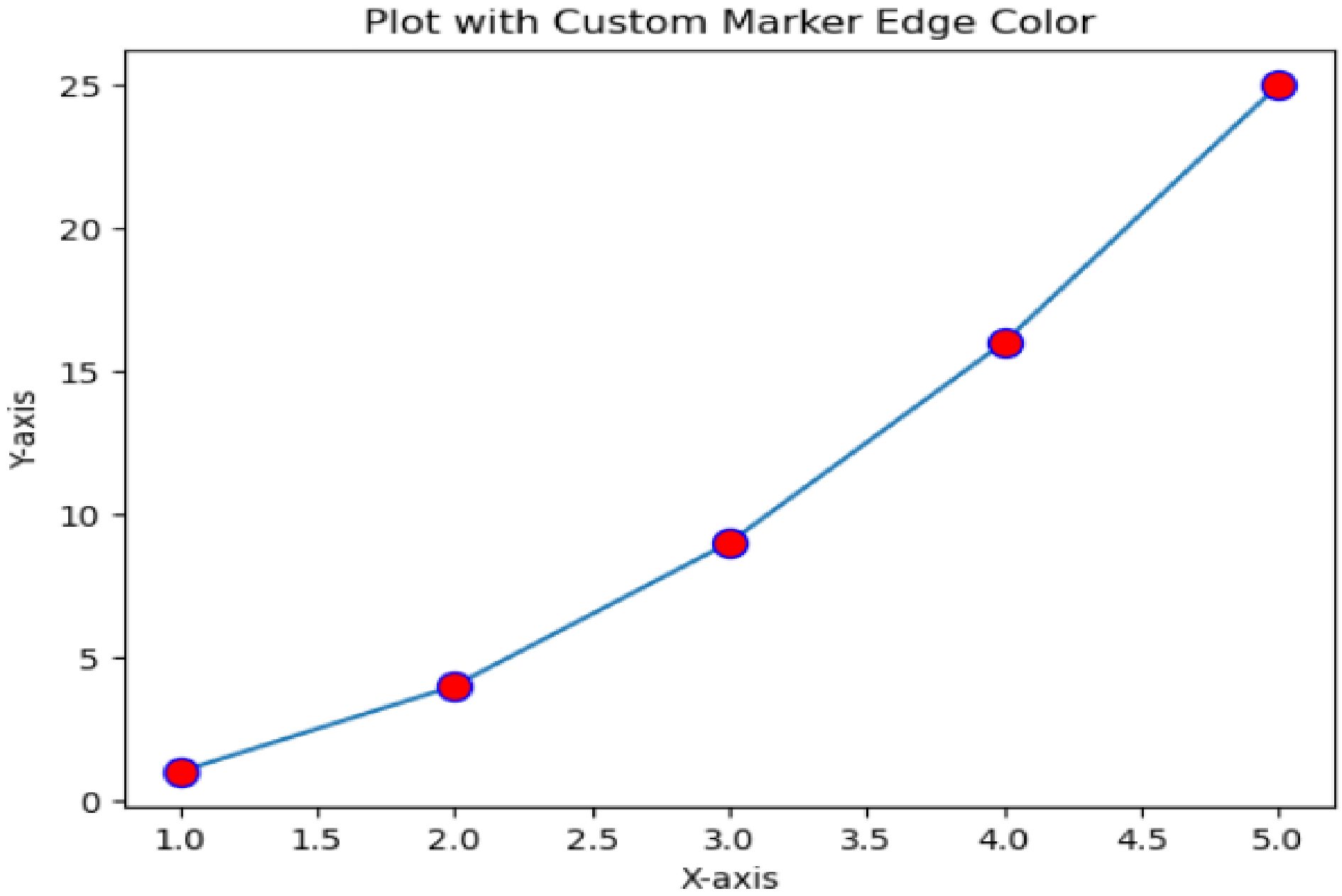
- `plt.show()`: Displays the plot on the screen.

Markers Face Size

You can use the keyword argument markerfacecolor or the shorter mfc to set the color of the edge of the markers:

[53]:

```python
import matplotlib.pyplot as plt

# Example data
x = [1, 2, 3, 4, 5]
y = [1, 4, 9, 16, 25]

# Plot with markers
plt.plot(x, y, marker='o', markerfacecolor='red', markeredgecolor='blue', markersize=10)

# Adding labels and title
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.title('Plot with Custom Marker Edge Color')

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



1. Imports:

   - `import matplotlib.pyplot as plt`: Imports the pyplot module from Matplotlib for plotting.

2. Example Data:

   - `x = [1, 2, 3, 4, 5]`: Defines the x-axis values.
   - `y = [1, 4, 9, 16, 25]`: Defines the y-axis values.

3. **Plotting:**

   - `plt.plot(x, y, marker='o', markerfacecolor='red', markeredgecolor='blue', markersize=10)`:
     - `marker='o'`: Specifies circular markers (`'o'`) at each data point.
     - `markerfacecolor='red'`: Sets the color of the marker faces to red.
     - `markeredgecolor='blue'`: Sets the color of the marker edges to blue.
     - `markersize=10`: Sets the size of the markers to 10 points.

4. **Adding Labels and Title:**

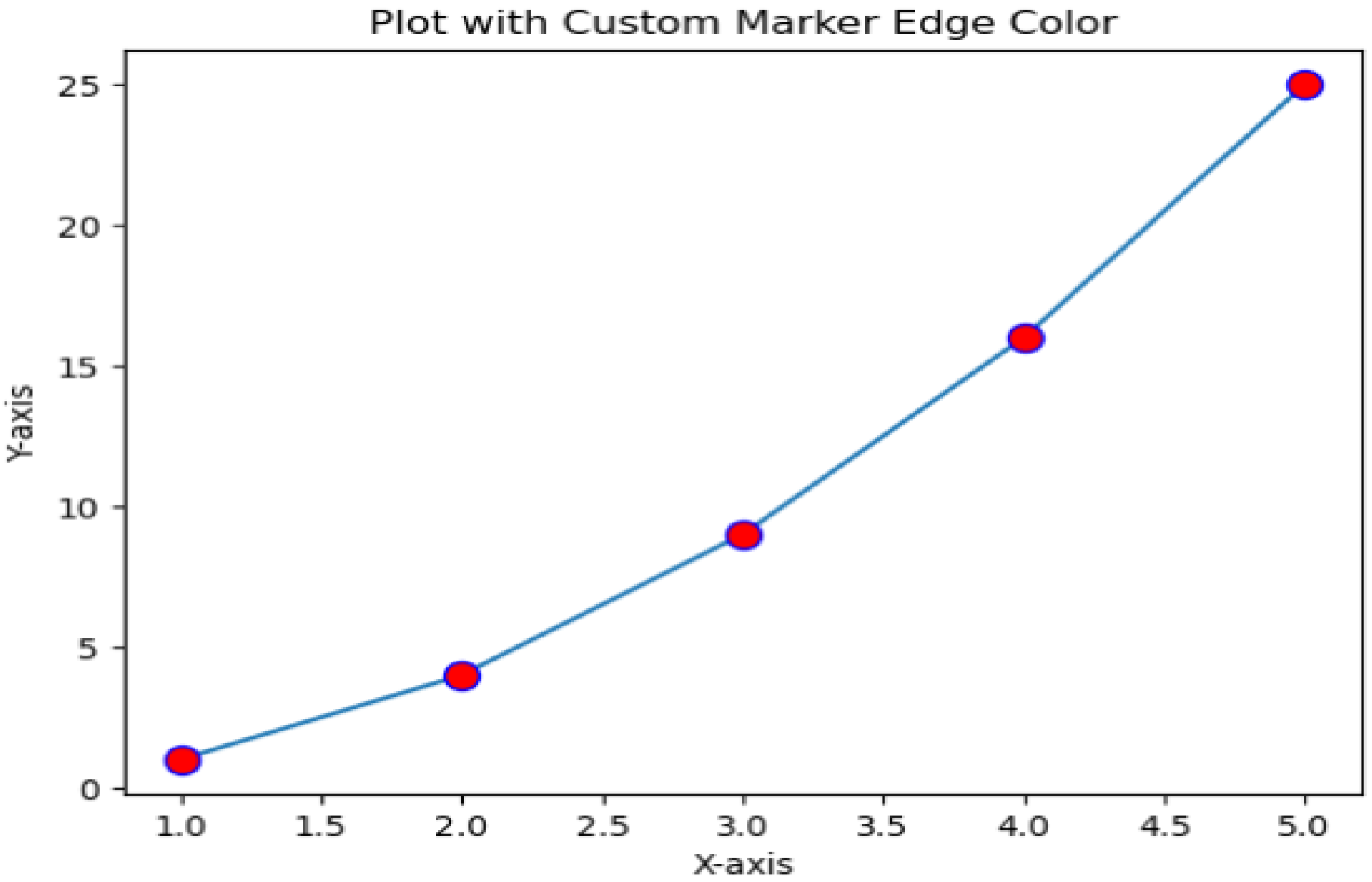
   - `plt.xlabel('X-axis')`: Adds a label to the x-axis.
   - `plt.ylabel('Y-axis')`: Adds a label to the y-axis.
   - `plt.title('Plot with Custom Marker Edge Color')`: Adds a title to the plot.

5. Displaying the Plot:

- `plt.show()`: Displays the plot on the screen.

[54]:

```
import matplotlib.pyplot as plt

# Example data
x = [1, 2, 3, 4, 5]
y = [1, 4, 9, 16, 25]

# Plot with markers
plt.plot(x, y, marker='o', mfc='red', mec='blue', markersize=10)

# Adding labels and title
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.title('Plot with Custom Marker Edge Color')

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

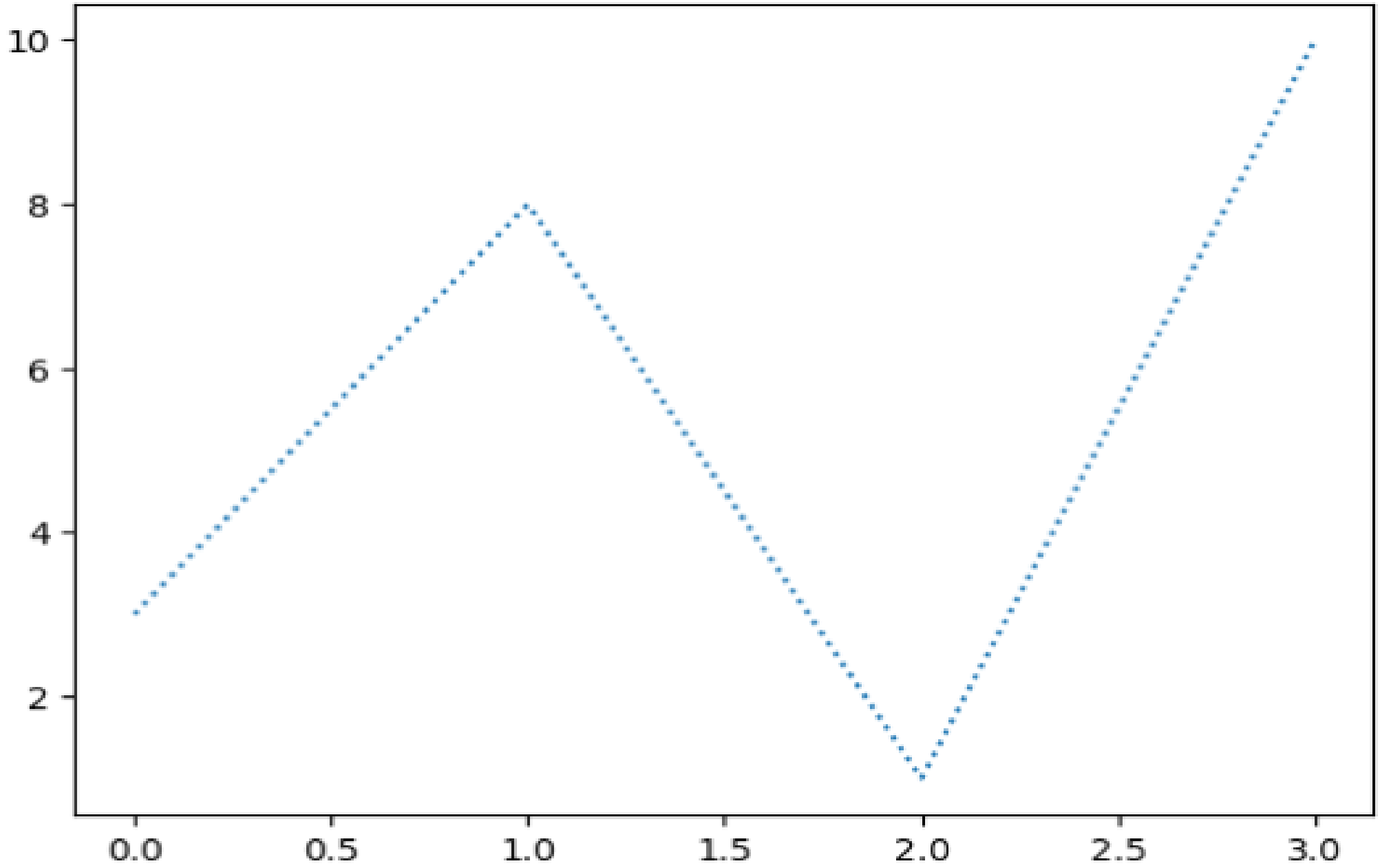


Linestyle Argument

You can use the keyword argument linestyle, or shorter ls, to change the style of the plotted line

[8]:

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

y=np.array([3,8,1,10])

plt.plot(y,linestyle='dotted')
plt.show()
```


10
8
6
4
2
0.0
0.5
1.0
1.5
2.0
2.5
3.0

1. Imports:

   - `import matplotlib.pyplot as plt`: Imports the pyplot module from Matplotlib for plotting.
   - `import numpy as np`: Imports the NumPy library for array operations.

2. Data Creation:

   - `y = np.array([3, 8, 1, 10])`: Defines an array `y` with four data points `[3, 8, 1, 10]`.

3. Plotting:

- `plt.plot(y, linestyle='dotted')`: Plots the values in the array `y`.
    - `linestyle='dotted'`: Specifies the line style as dotted (`':'`).
        - This means the plot will connect the points `(0, 3)`, `(1, 8)`, `(2, 1)`, and `(3, 10)` with dotted lines.

4. Displaying the Plot:

- `plt.show()`: Displays the plot on the screen.

Line Reference

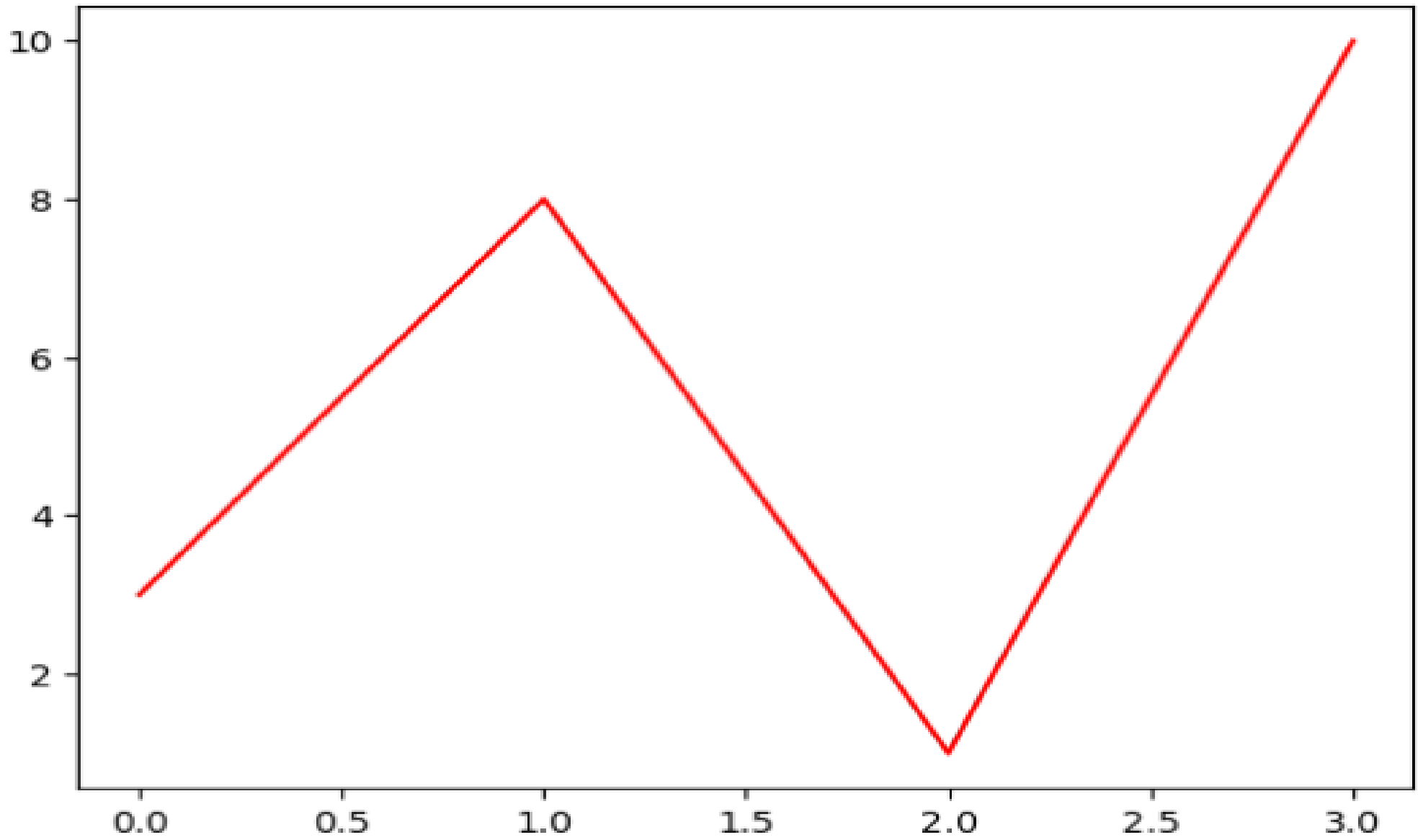
ls Syntax	linestyle Syntax
'-'	'solid'
':'	'dotted'
'--'	'dashed'
'-.'	'dashdot'

Line Color Argument

You can use the keyword argument color or the shorter c to set the color of the line:

[9]:

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


y=np.array([3,8,1,10])


plt.plot(y,color='red')
plt.show()
```


10
8
6
4
2
0.0
0.5
1.0
1.5
2.0
2.5
3.0

1. Imports:

   - `import matplotlib.pyplot as plt`: Imports the pyplot module from Matplotlib for plotting.
   - `import numpy as np`: Imports the NumPy library for array operations.

2. Data Creation:

   - `y = np.array([3, 8, 1, 10])`: Defines an array `y` with four data points `[3, 8, 1, 10]`.

3. Plotting:

- `plt.plot(y, color='red')`: Plots the values in the array `y`.
    - `color='red'`: Specifies the color of the line as red (`'red'`).
        - This means the plot will connect the points `(0, 3)`, `(1, 8)`, `(2, 1)`, and `(3, 10)` with a red line.

4. Displaying the Plot:

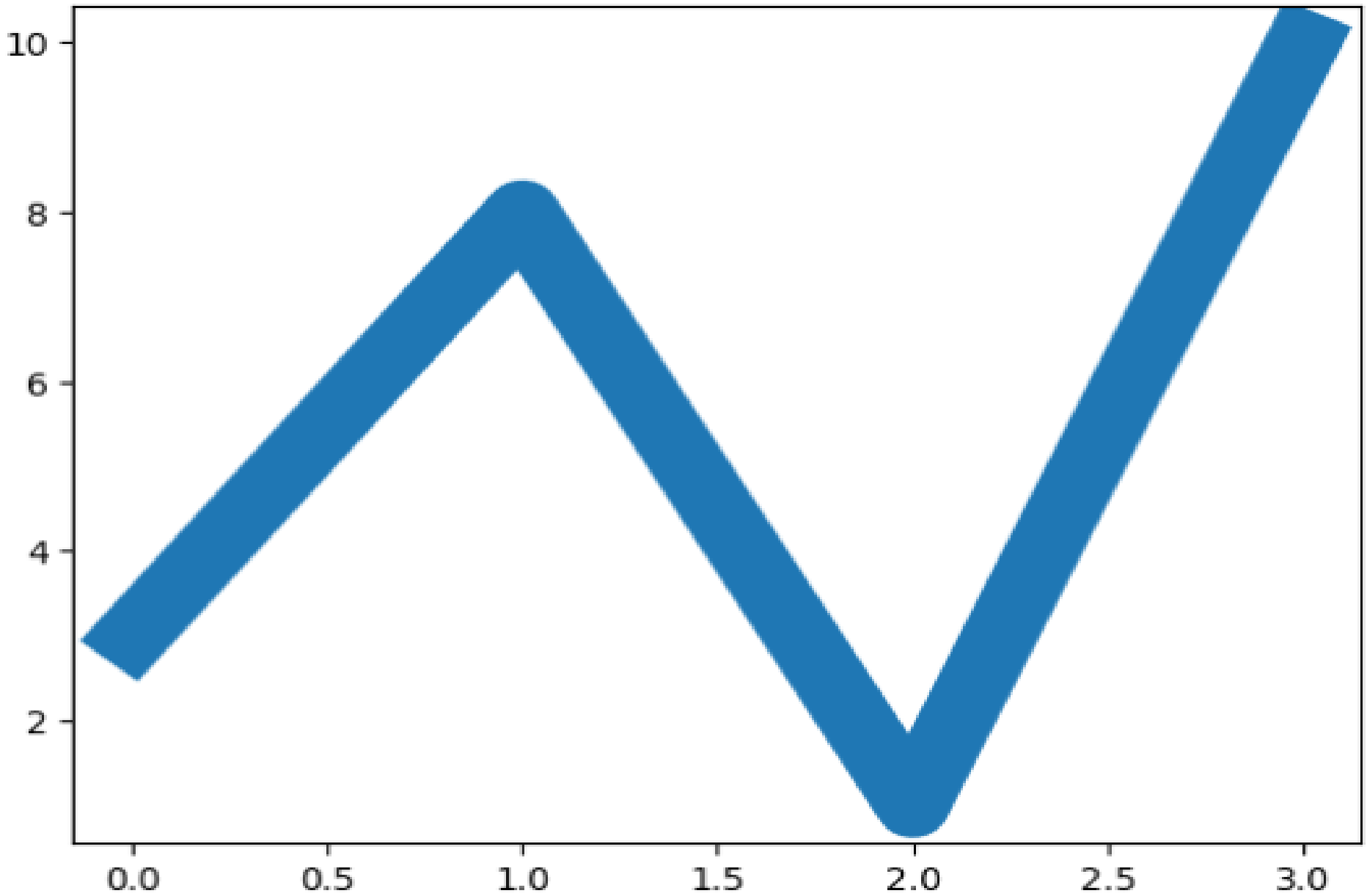
- `plt.show()`: Displays the plot on the screen.

Line width Argument

You can use the keyword argument linewidth or the shorter lw to change the width of the line

[10]:

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

y=np.array([3,8,1,10])

plt.plot(y,linewidth='20.6')
plt.show()
```


10
8
6
4
2
0.0
0.5
1.0
1.5
2.0
2.5
3.0

1. Importing Libraries:

   - `import matplotlib.pyplot as plt`: Imports the `matplotlib.pyplot` module under the alias `plt`, which is a popular plotting library in Python.

2. Creating Data:

   - `import numpy as np`: Imports the `numpy` library under the alias `np`, which is used for numerical operations in Python.
   - `y = np.array([3, 8, 1, 10])`: Creates a NumPy array `y` containing the values `[3, 8, 1, 10]`. These values represent the y-coordinates of points to be plotted.

3. Plotting Data:

- `plt.plot(y, linewidth=20.6)`: Plots the values in `y` as a line plot. The `linewidth` parameter specifies the width of the line to be `20.6` points. This parameter should be a numerical value.

4. Displaying the Plot:

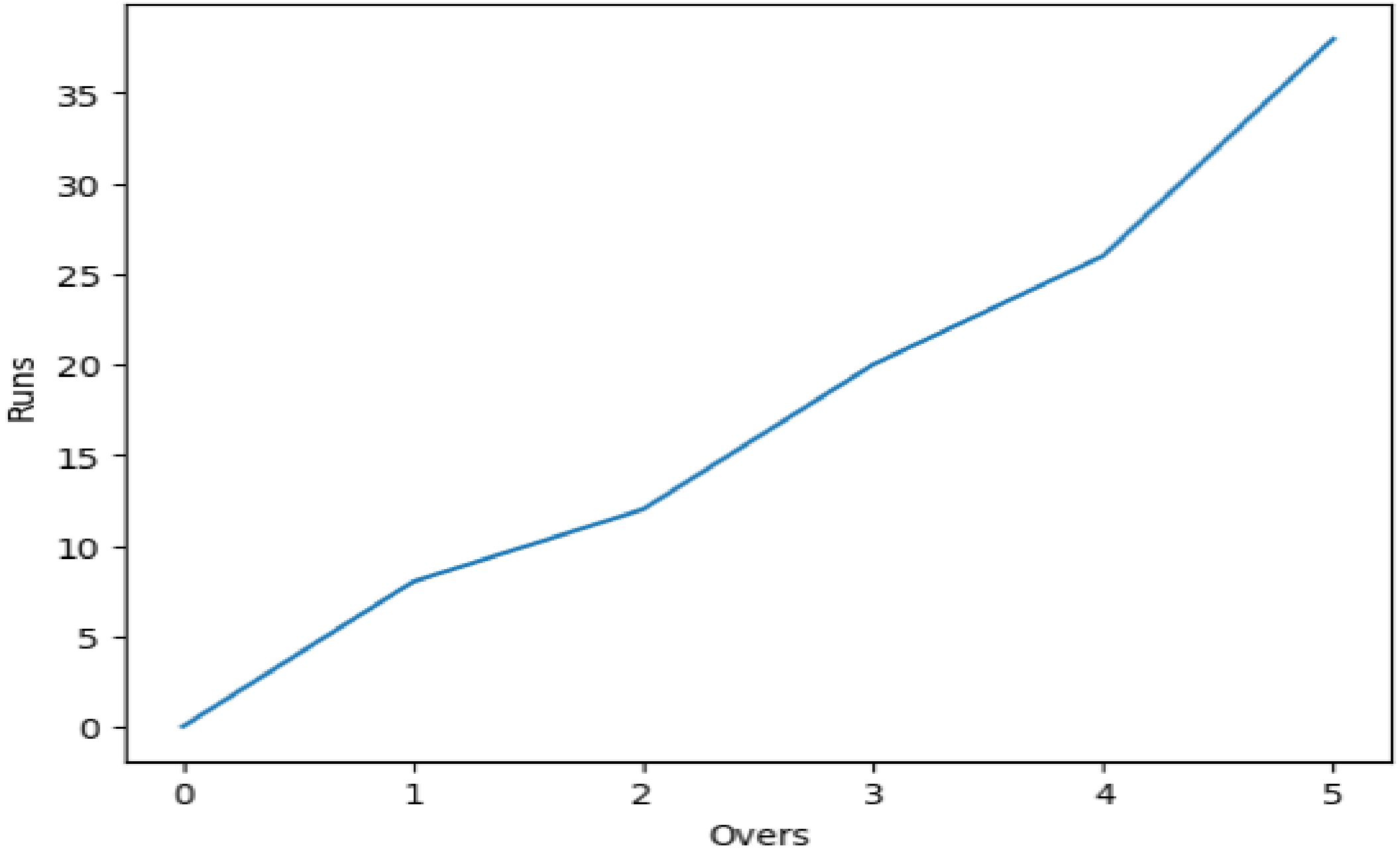
- `plt.show()`: Displays the plot on the screen. This function is necessary to show the plot after it has been configured with all the desired elements.

Create Labels

With Pyplot, you can use the xlabel() and ylabel() functions to set a label for the x- and y-axis.

[11]:

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

x=np.array([0,1,2,3,4,5])
y=np.array([0,8,12,20,26,38])

plt.plot(x,y)
plt.xlabel("Overs")
plt.ylabel("Runs")
plt.show()
```


Runs
0
5
10
15
20
25
30
35
0
1
2
3
4
5
Overs

Code Summary

1. **Importing Libraries:**
    - `` `import matplotlib.pyplot as plt` ``: Imports the `` `matplotlib.pyplot` `` module under the alias `` `plt` ``, which is used for creating plots.
    - `` `import numpy as np` ``: Imports the `` `numpy` `` library under the alias `` `np` ``, which is used for numerical operations.
2. **Creating Data:**
    - `` `x = np.array([0, 1, 2, 3, 4, 5])` ``: Creates a NumPy array `` `x` `` containing the x-coordinates of the data points.
    - `` `y = np.array([0, 8, 12, 20, 26, 38])` ``: Creates a NumPy array `` `y` `` containing the y-coordinates of the data points.

3. **Plotting Data:**

    - `plt.plot(x, y)`: Plots `y` against `x` as a line plot. Matplotlib connects the points `(x[i], y[i])` with lines.

4. **Adding Labels:**

    - `plt.xlabel("Overs")`: Adds a label to the x-axis with the text `"Overs"`.
    - `plt.ylabel("Runs")`: Adds a label to the y-axis with the text `"Runs"`.

5. **Displaying the Plot:**

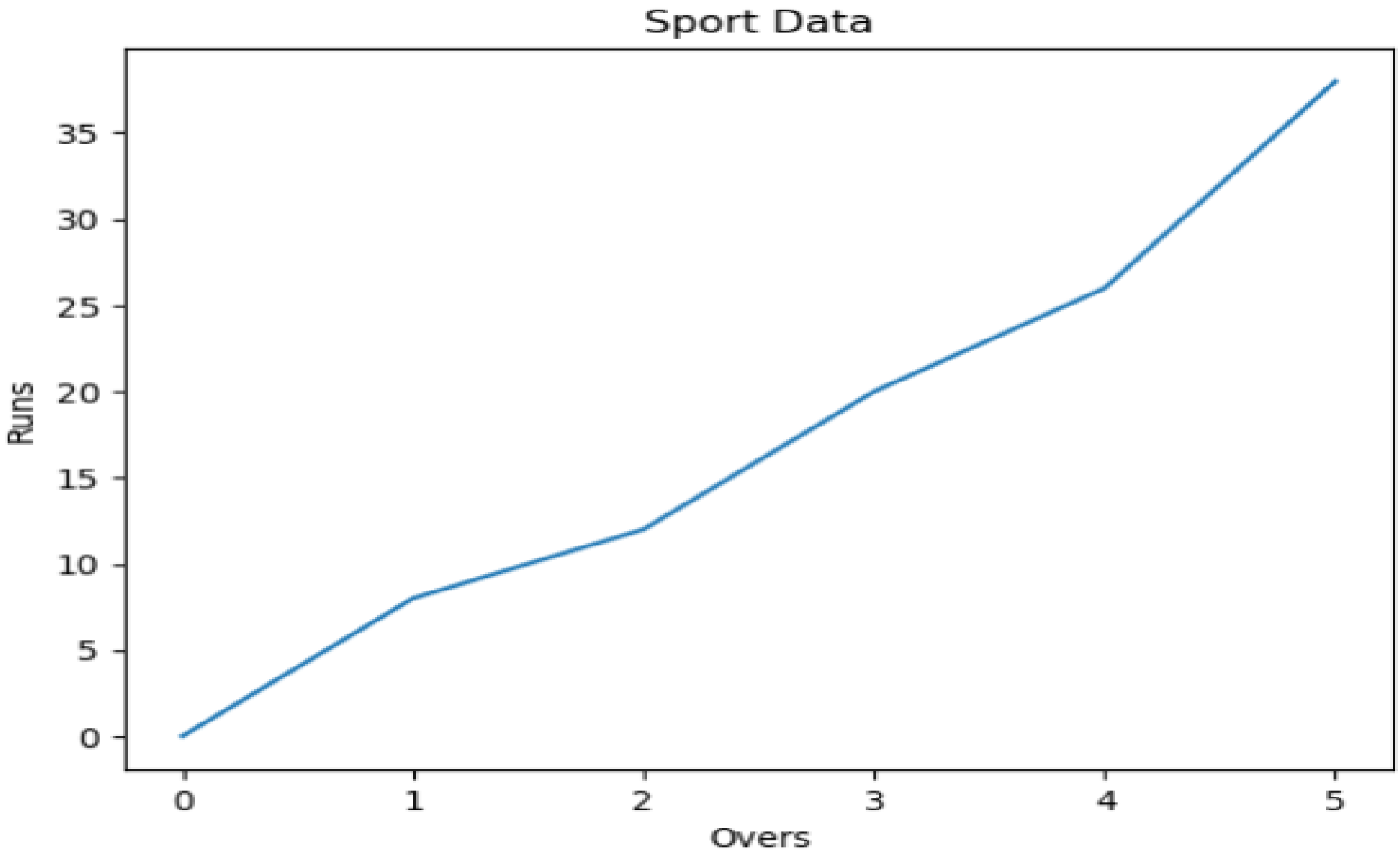
    - `plt.show()`: Displays the plot on the screen. This function is necessary to show the plot after configuring it with labels and other elements.

Create Title

With Pyplot, you can use the title() function to set a title for the plot.

[12]:

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

x=np.array([0,1,2,3,4,5])
y=np.array([0,8,12,20,26,38])

plt.plot(x,y)
plt.xlabel("Overs")
plt.ylabel("Runs")
plt.title("Sport Data")
plt.show()
```

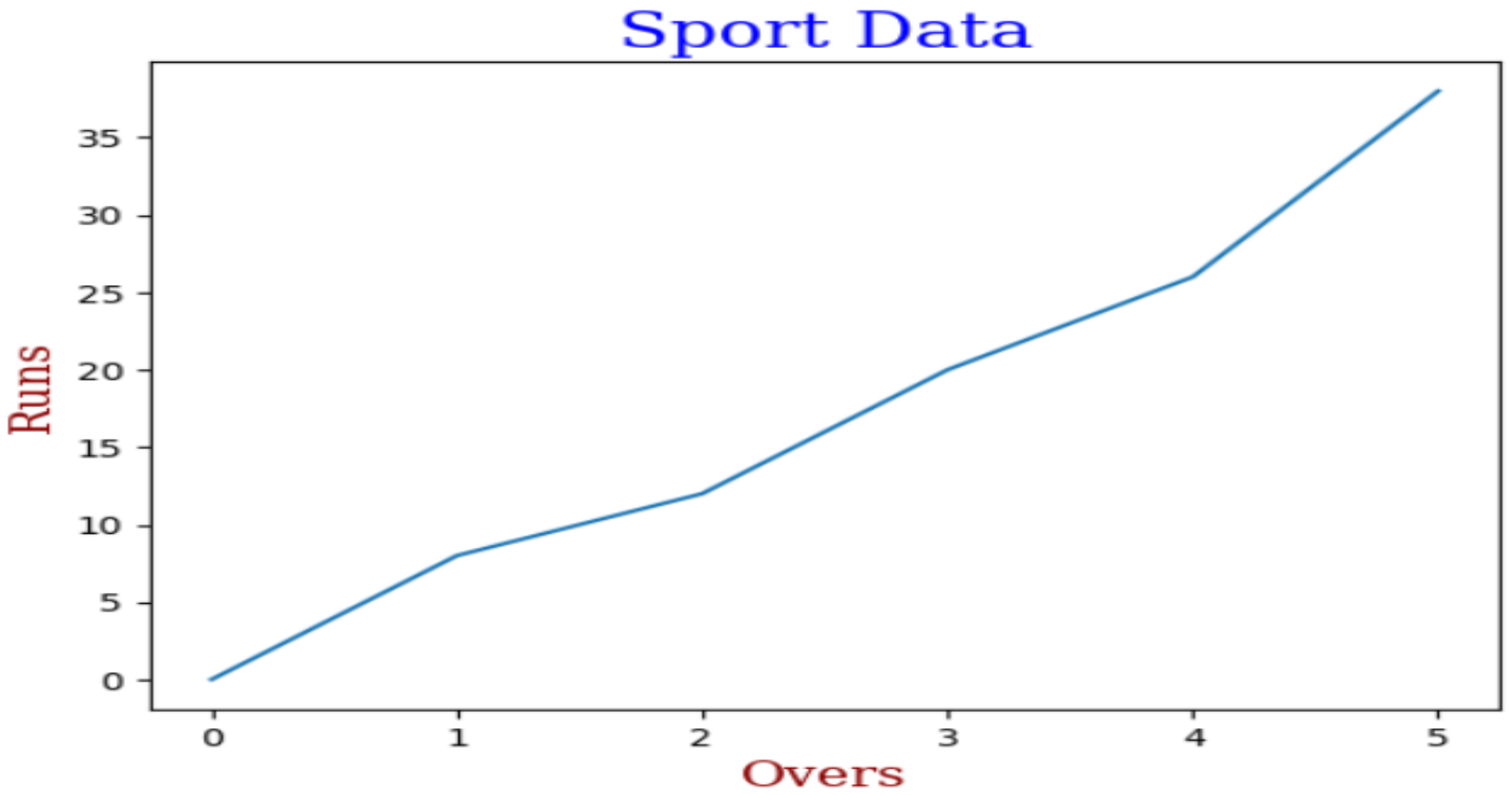

Sport Data
Runs
Overs
0
5
10
15
20
25
30
35
0
1
2
3
4
5

Set Font Properties for Title and Labels

You can use the fontdict parameter in xlabel(), ylabel(), and title() to set font properties for the title and labels.

In [26]:
```
import matplotlib.pyplot as plt
import numpy as np
x = np.array([0,1,2,3,4,5])
y = np.array([0,8,12,20,26,38])
font1 = {'family':'serif','color':'blue','size':20}
font2 = {'family':'serif','color':'darkred','size':15}
plt.plot(x,y)
plt.xlabel("Overs",fontdict=font2)
plt.ylabel("Runs" ,fontdict=font2)
plt.title("Sport Data" ,fontdict=font1)
plt.show()
```

Set Font Properties for Title and Labels



Summary of the Code

1. Imports:
    - `matplotlib.pyplot` is imported as `plt` for plotting functionalities.
    - `numpy` is imported as `np` for numerical operations.
2. Data Preparation:
    - Two NumPy arrays `x` and `y` are created:
        - `x` contains values `[0, 1, 2, 3, 4, 5]` representing overs.
        - `y` contains values `[0, 8, 12, 20, 26, 38]` representing runs scored.

3. **Font Dictionaries:**

   - Two dictionaries are defined to customize font properties:
     - `font1`: Specifies serif font in blue color with size 20.
     - `font2`: Specifies serif font in dark red color with size 15.

4. **Plotting:**

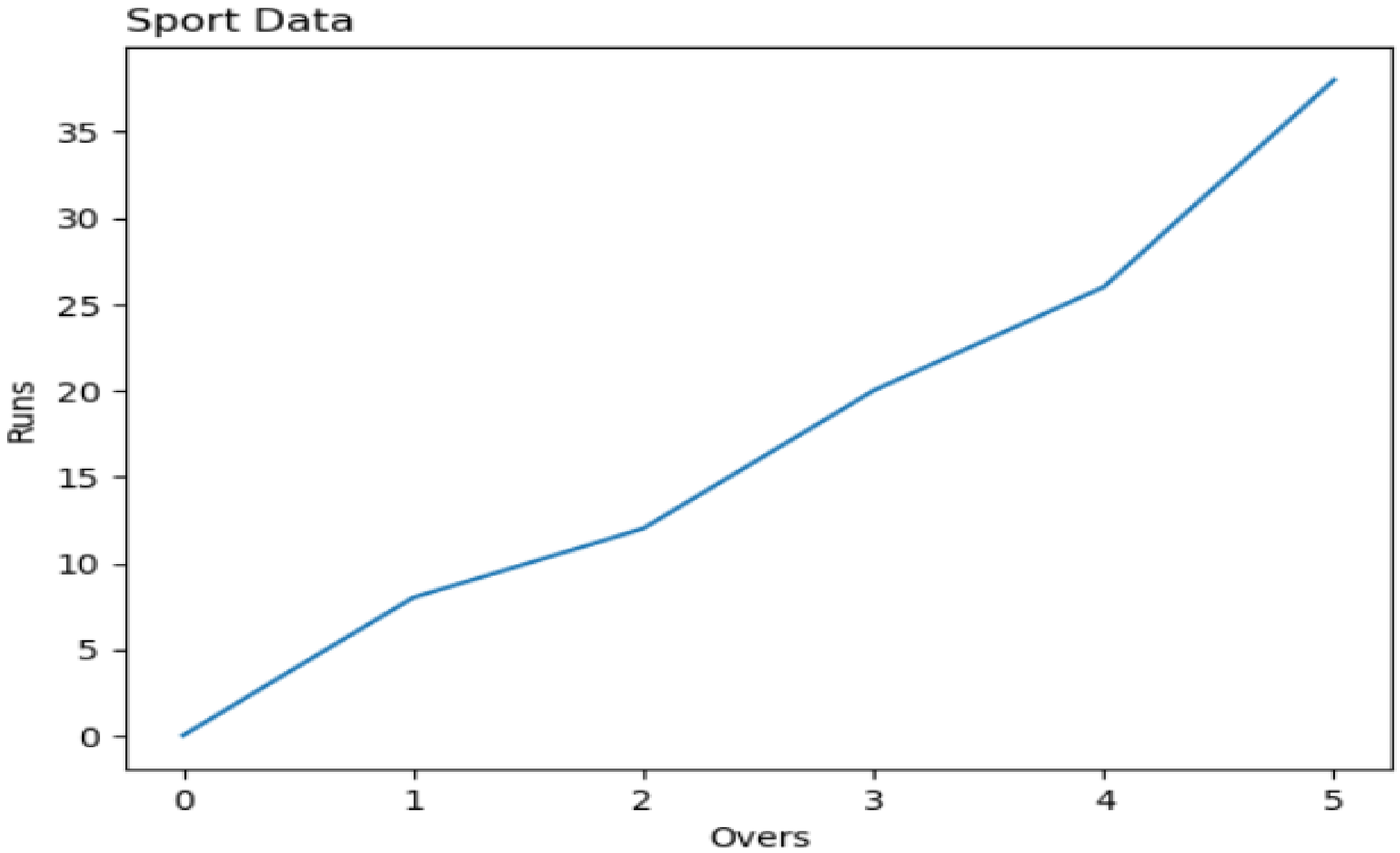
   - `plt.plot(x, y)`: Plots a line graph with `x` on the x-axis and `y` on the y-axis.
   - `plt.xlabel("Overs", fontdict=font1)`: Sets the x-axis label to "Overs" using `font1`.
   - `plt.ylabel("Runs", fontdict=font2)`: Sets the y-axis label to "Runs" using `font2`.
   - `plt.title("Sport Data", fontdict=font1)`: Sets the title of the plot to "Sport Data" using `font1`.
   - `plt.show()`: Displays the plot on screen.

Position the Title

- You can use the loc parameter in title() to position the title.
- Legal values are: 'left', 'right', and 'center'.
- **Default value is 'center'.**

In [27]:

```
import matplotlib.pyplot as plt
import numpy as np
x = np.array([0,1,2,3,4,5])
y = np.array([0,8,12,20,26,38])
plt.plot(x,y)
plt.xlabel("Overs")
plt.ylabel("Runs")
plt.title("Sport Data", loc="left")
plt.show()
```

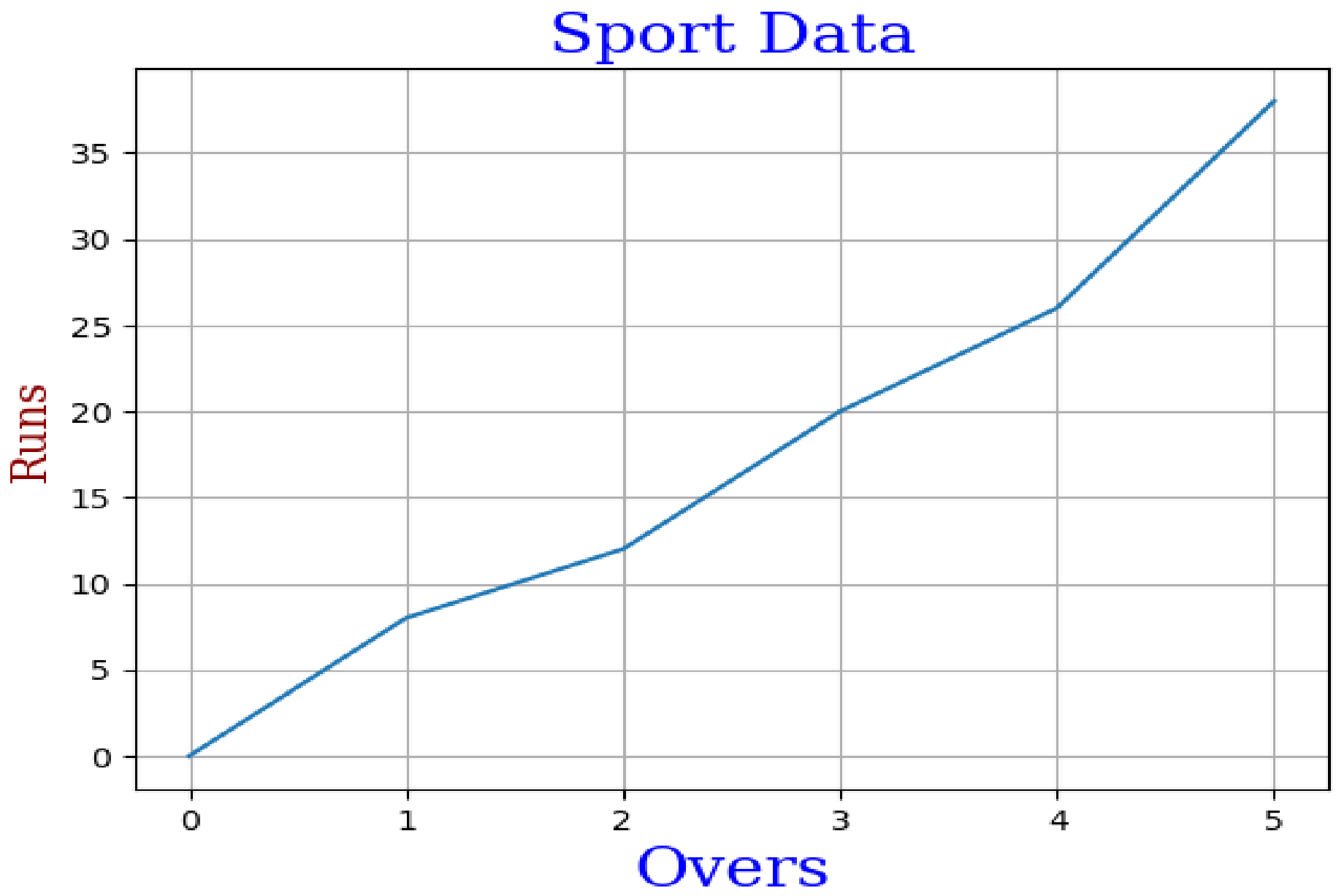

Sport Data
Runs
Overs
0
5
10
15
20
25
30
35
0
1
2
3
4
5

Add Grid Lines to a Plot

With Pyplot, you can use the grid() function to add grid lines to the plot.

[19]:

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

x=np.array([0,1,2,3,4,5])
y=np.array([0,8,12,20,26,38])
font1={'family':'serif','color':'blue','size':20}
font2={'family':'serif','color':'darkred','size':15}
plt.plot(x,y)
plt.xlabel("Overs",fontdict=font1)
plt.ylabel("Runs",fontdict=font2)
plt.title("Sport Data",fontdict=font1)
plt.grid(True)
plt.show()
```


Sport Data
Runs
Overs
0
5
10
15
20
25
30
35
0
1
2
3
4
5

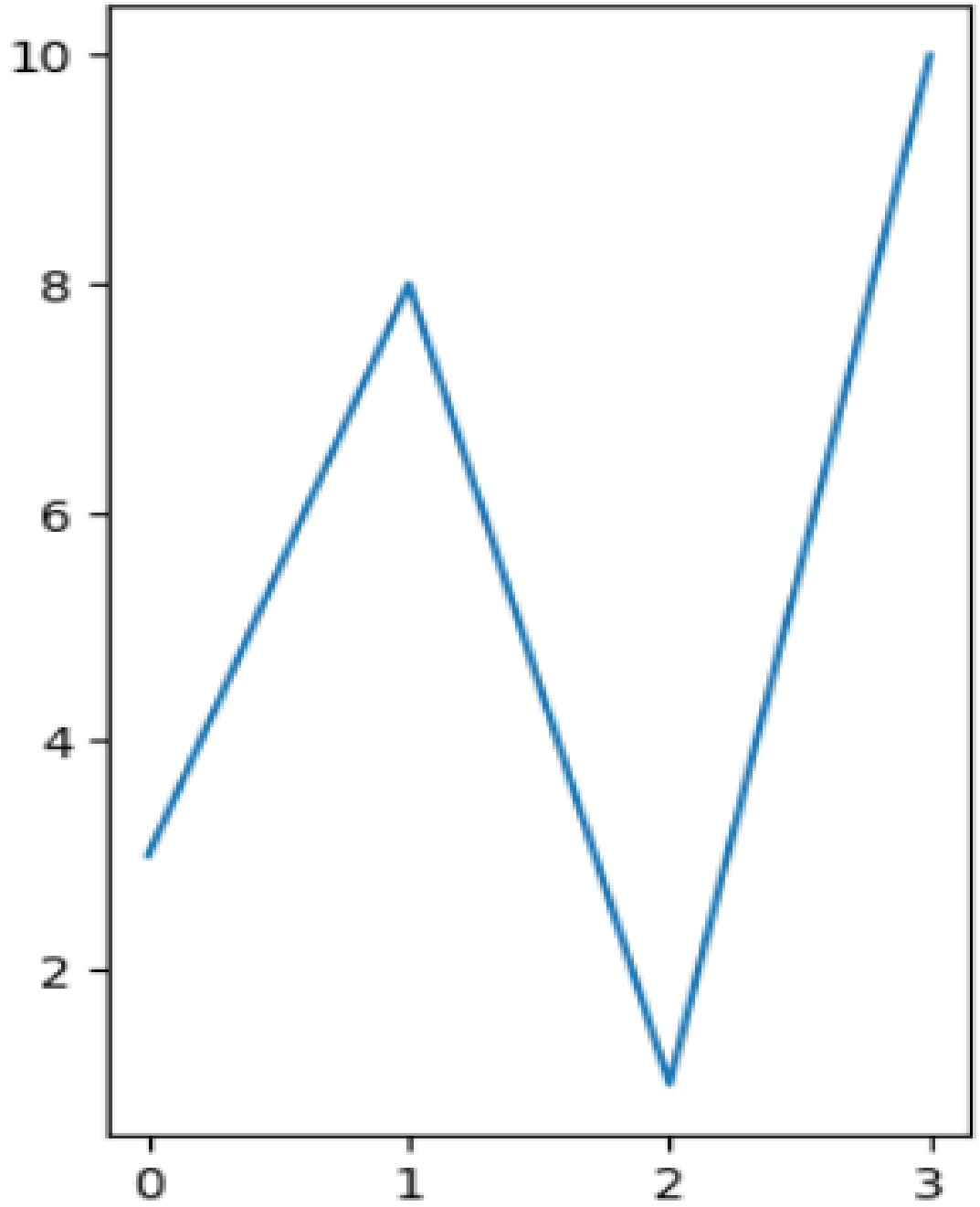
Display Multiple Plots

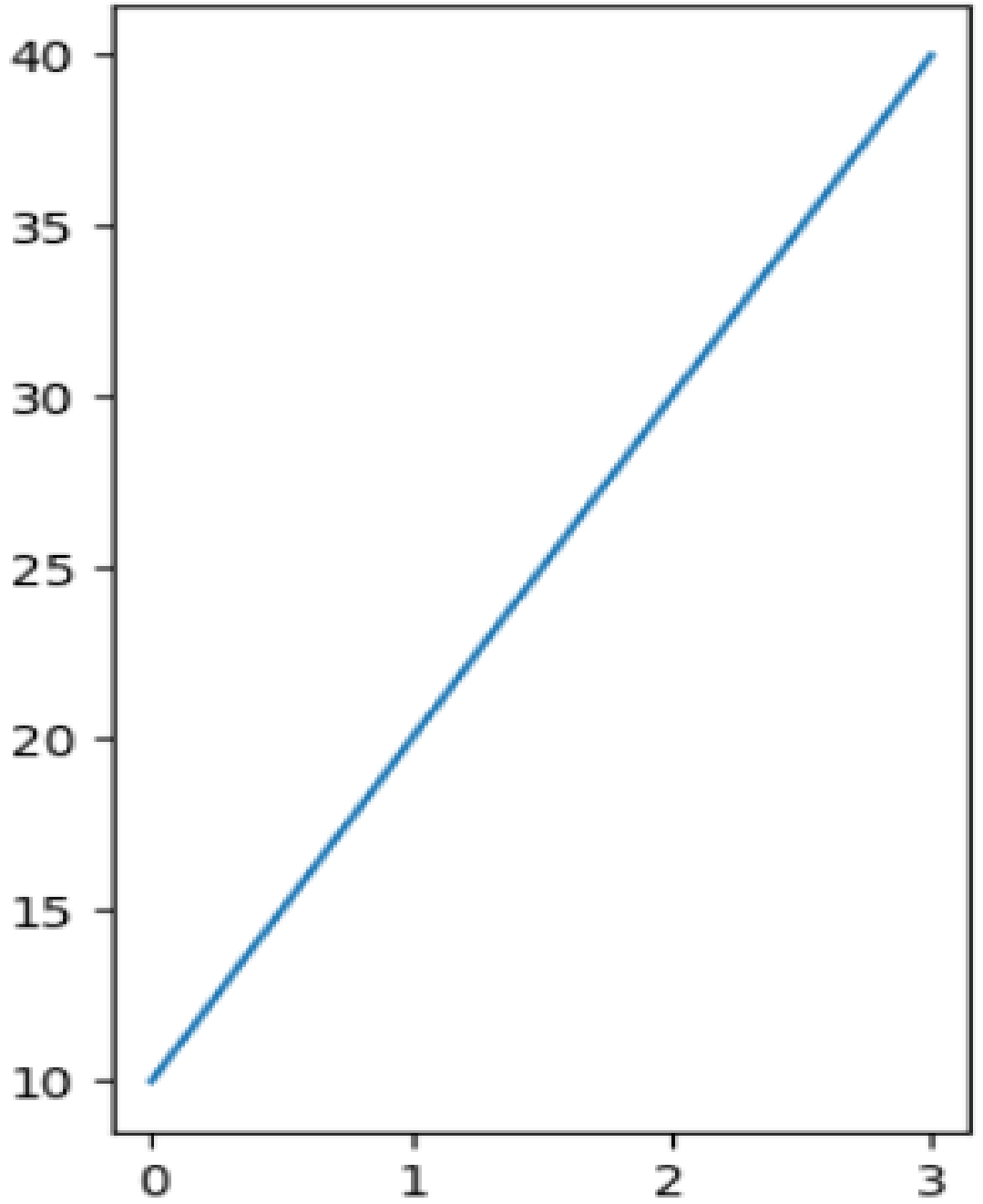
- With the subplot() function you can draw multiple plots in one figure
- The subplot() function takes three arguments that describes the layout of the figure.
- The layout is organized in rows and columns, which are represented by the first and second argument.
- The third argument represents the index of the current plot.

Display Multiple Plots

In [32]:

```
import matplotlib.pyplot as plt
import numpy as np
#plot 1
x = np.array([0,1,2,3])
y = np.array([3,8,1,10])
plt.subplot(1,2,1)
plt.plot(x,y)
#plot 2
x = np.array([0,1,2,3])
y = np.array([10,20,30,40])
plt.subplot(1,2,2)
plt.plot(x,y)
plt.show()
```


10
8
6
4
2
0
1
2
3


40
35
30
25
20
15
10
0
1
2
3

Explanation:

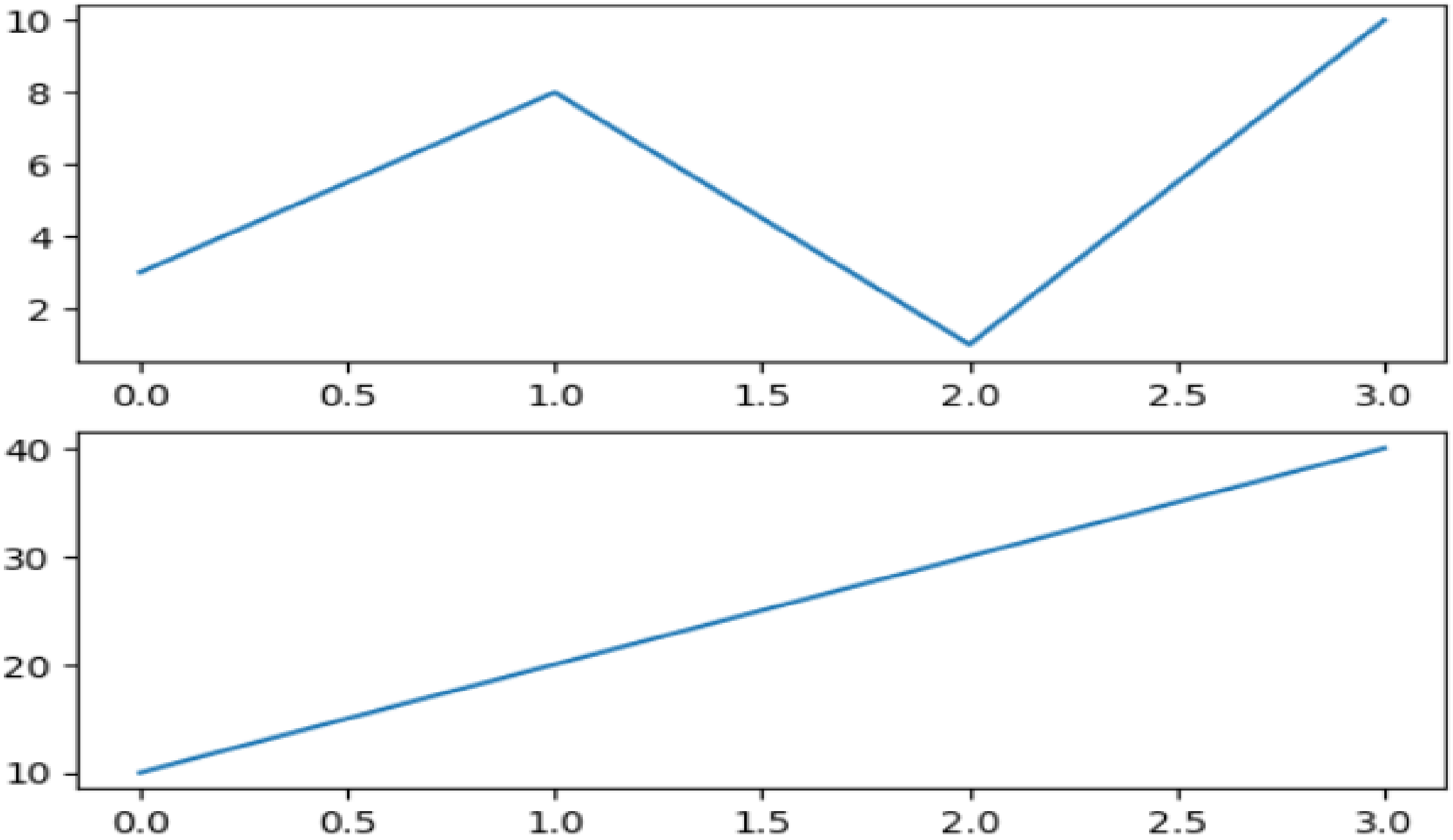
- `plt.subplot(1,2,1)` and `plt.subplot(1,2,2)` are used to divide the plotting area into a grid of 1 row and 2 columns, indicating that you want two subplots side by side.
- `plt.plot(x, y)` is used to plot the data points defined by `x` and `y` arrays in each subplot.
- The first subplot (`subplot(1,2,1)`) shows a plot where `y` ranges from 3 to 10 across the points defined by `x`.
- The second subplot (`subplot(1,2,2)`) shows a plot where `y` ranges from 10 to 40 across the points defined by `x`.

When you run this code, you should see a figure window pop up with two plots side by side, each displaying the respective data points specified in `x` and `y`.

Display Multiple Plots

In [33]:

```
import matplotlib.pyplot as plt
import numpy as np
#plot 1
x = np.array([0,1,2,3])
y = np.array([3,8,1,10])
plt.subplot(2,1,1)
plt.plot(x,y)
#plot 2
x = np.array([0,1,2,3])
y = np.array([10,20,30,40])
plt.subplot(2,1,2)
plt.plot(x,y)
plt.show()
```


10
8
6
4
2
0.0
0.5
1.0
1.5
2.0
2.5
3.0
40
30
20
10
0.0
0.5
1.0
1.5
2.0
2.5
3.0

Explanation:

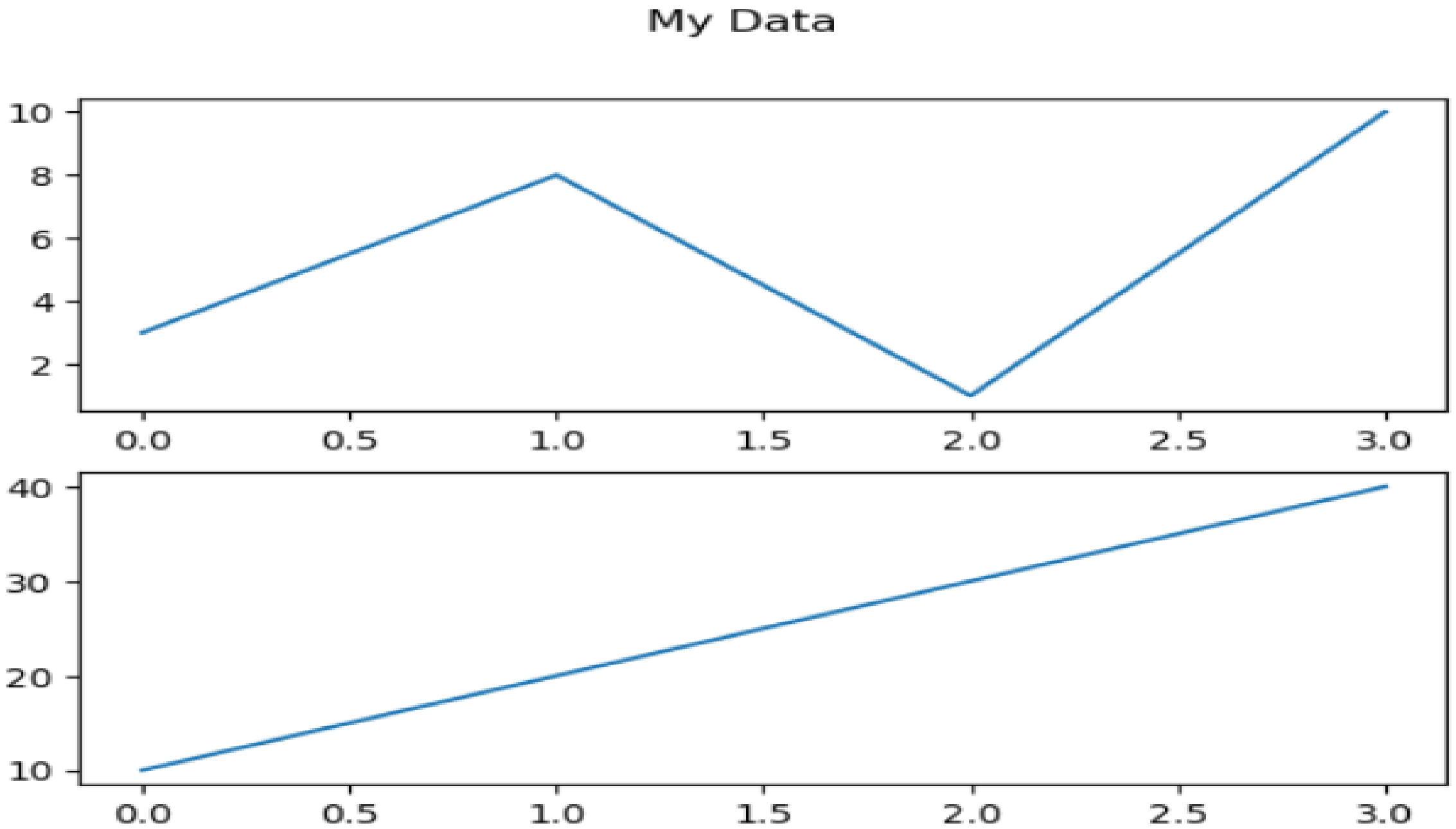
- `plt.subplot(2,1,1)` and `plt.subplot(2,1,2)` are used to divide the plotting area into a grid of 2 rows and 1 column, indicating that you want two subplots stacked vertically.
- `plt.plot(x, y)` is used to plot the data points defined by `x` and `y` arrays in each subplot.
- The first subplot (`subplot(2,1,1)`) shows a plot where `y` ranges from 3 to 10 across the points defined by `x`.
- The second subplot (`subplot(2,1,2)`) shows a plot where `y` ranges from 10 to 40 across the points defined by `x`.

Super Title

You can add a title to the entire figure with the suptitle() function:

In [37]:

```
import matplotlib.pyplot as plt
import numpy as np
#plot 1
x = np.array([0,1,2,3])
y = np.array([3,8,1,10])
plt.subplot(2,1,1)
plt.plot(x,y)
#plot 2
x = np.array([0,1,2,3])
y = np.array([10,20,30,40])
plt.subplot(2,1,2)
plt.plot(x,y)
plt.suptitle("My Data")
plt.show()
```



Creating Scatter Plots

- With Pyplot, you can use the **scatter**() function to draw a scatter plot.
- The scatter() function plots **one dot for each observation**.
- It needs t**wo arrays of the same length**, one for the values of the x-axis, and one for values on the y-axis:

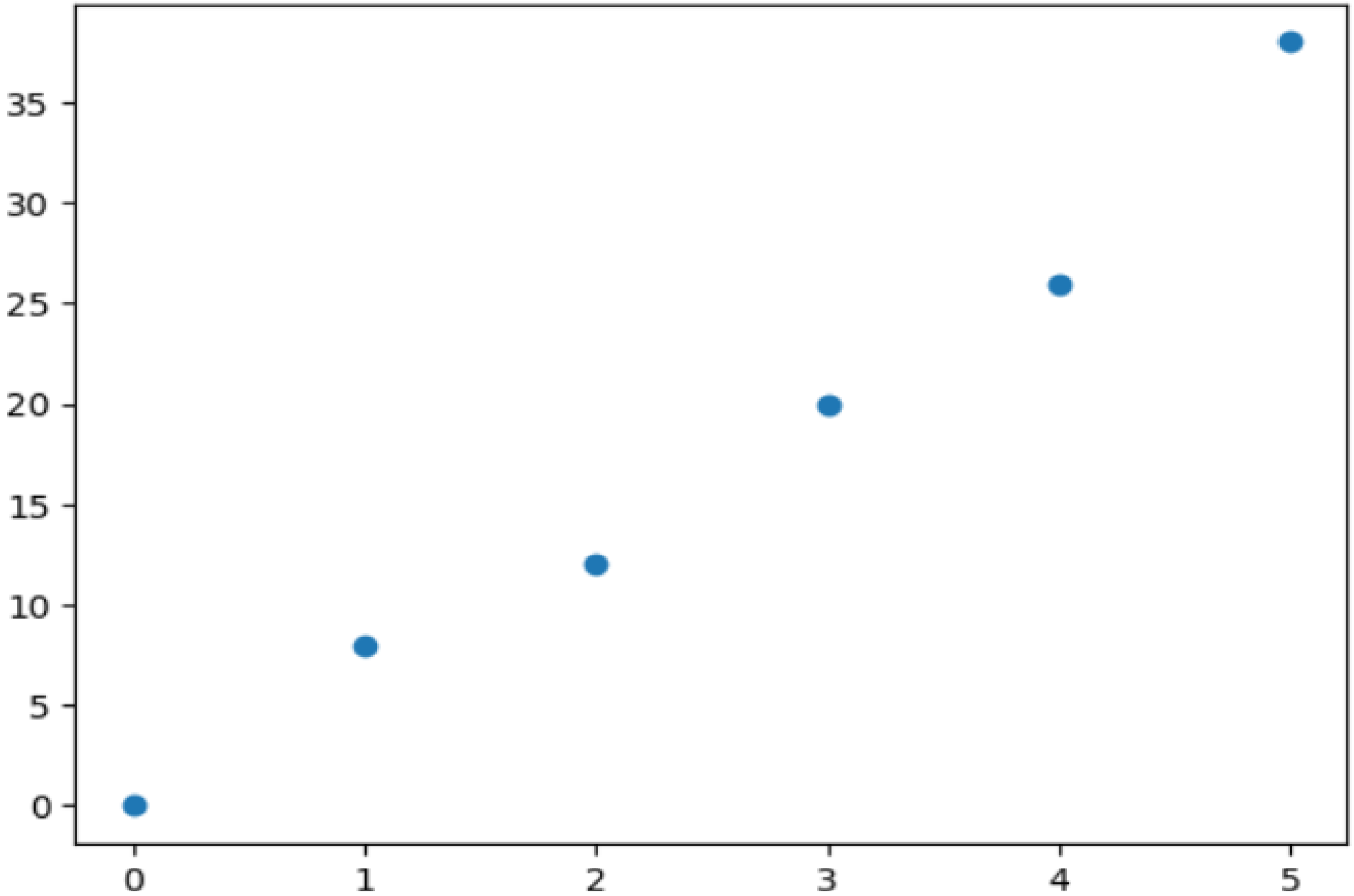
2. **Scatter Plot:**

- **Representation:** A scatter plot displays individual data points as markers without connecting them.
- **Usage:** Useful for visualizing the relationship between two variables, especially when examining correlation or distribution patterns.
- **Data Type:** Suitable for both continuous and categorical data.
- **Interpretation:** Helps in identifying clusters, trends, or outliers in the data distribution.
- **Example:** Plotting heights and weights of individuals, examining the relationship between study hours and exam scores, or analyzing the spread of data points.

Scatter Plots

[20]:

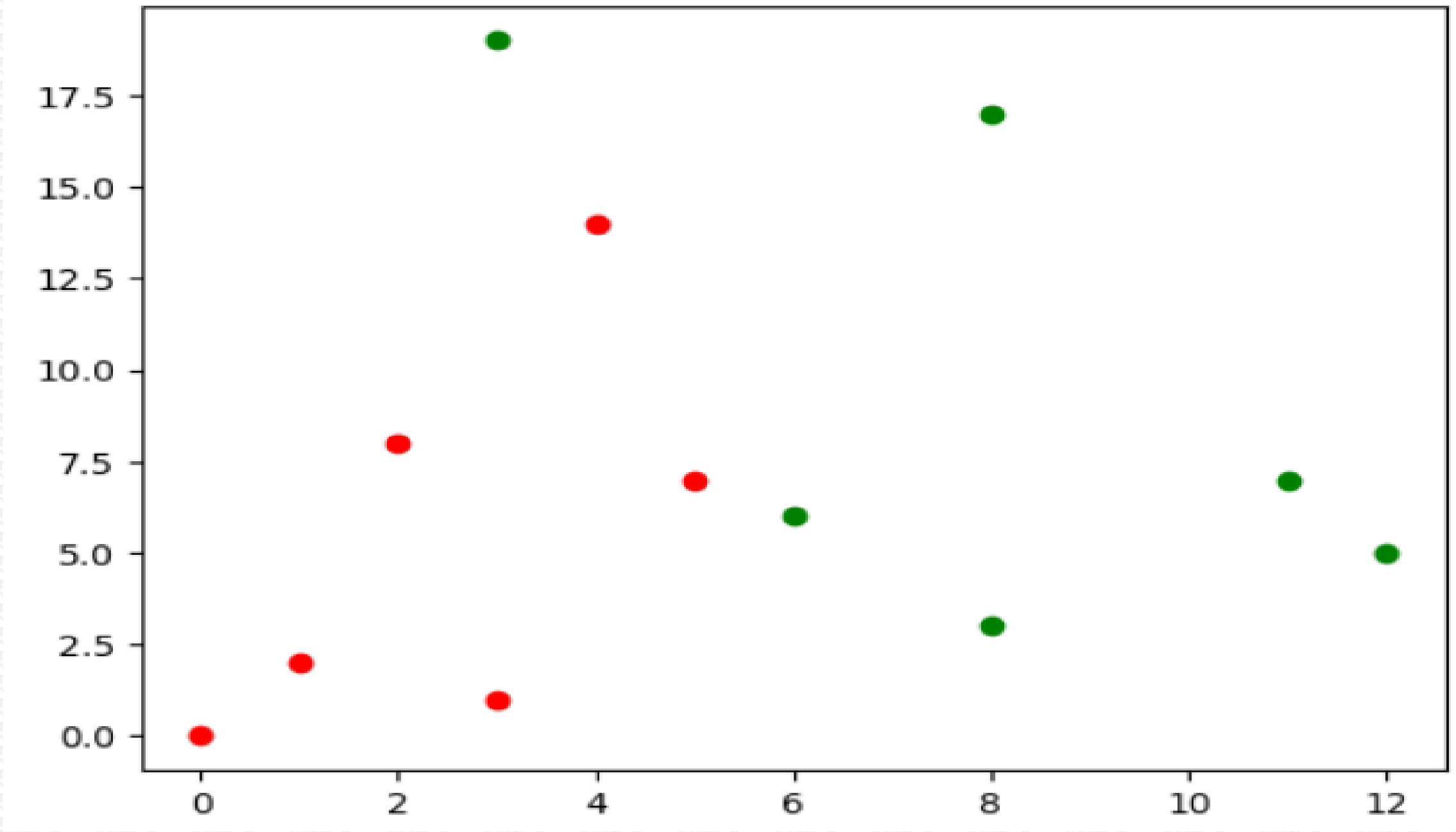
```
import matplotlib.pyplot as plt
import numpy as np


x=np.array([0,1,2,3,4,5])
y=np.array([0,8,12,20,26,38])
plt.scatter(x,y)
plt.show()
```


35
30
25
20
15
10
5
0
0
1
2
3
4
5

Compare Plots

In [39]:

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

x = np.array([0,1,2,3,4,5])
y = np.array([0,2,8,1,14,7])
plt.scatter(x,y,color='red')

x = np.array([12,6,8,11,8,3])
y = np.array([5,6,3,7,17,19])
plt.scatter(x,y,color='green')
plt.show()
```



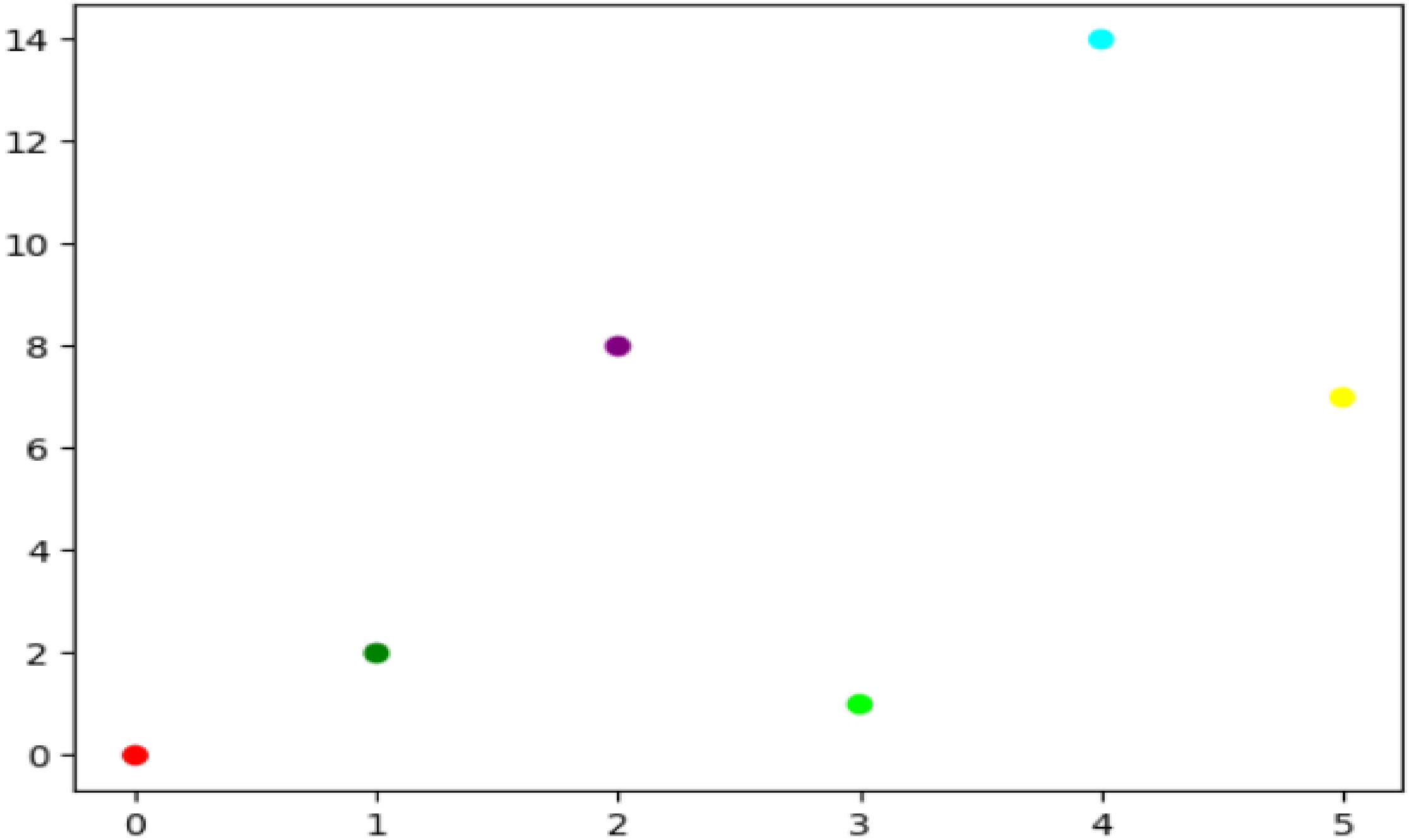
Code Summary:

- The code uses Matplotlib (`plt`) for plotting.
- Two sets of data points (`x` and `y` arrays) are defined using NumPy arrays.
- Two scatter plots are created, one with red points and the other with green points.
- Finally, the combined plot is displayed using `plt.show()`.

This script will produce a single plot with two sets of scattered points, differentiated by color.

Color each dots

In [40]:
```
import matplotlib.pyplot as plt
import numpy as np

x = np.array([0,1,2,3,4,5])
y = np.array([0,2,8,1,14,7])
mycolor=['red','green','purple','lime','aqua','yellow']

plt.scatter(x,y,color= mycolor )

plt.show()
```


14
12
10
8
6
4
2
0
0
1
2
3
4
5

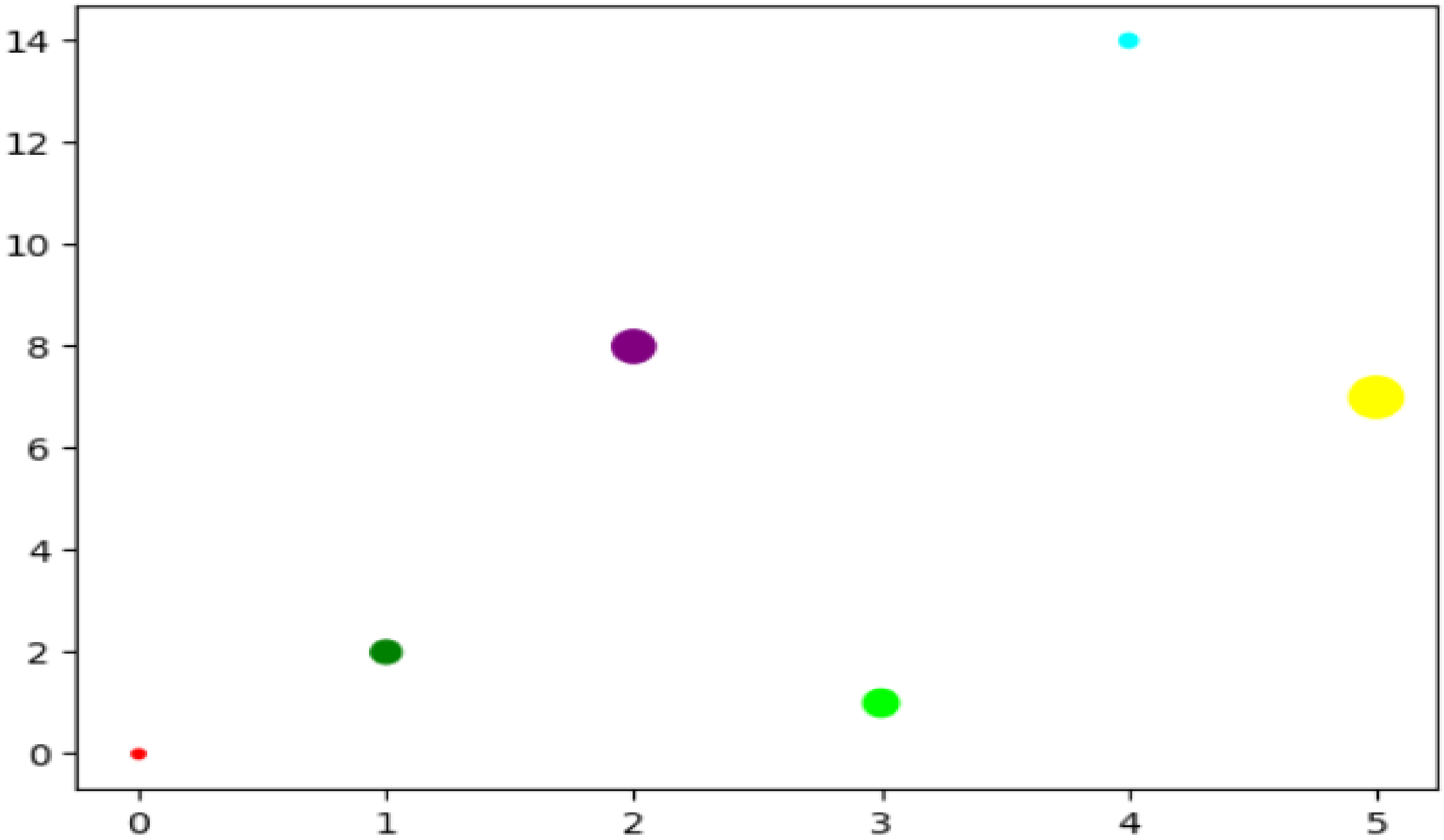
Code Summary:

- The code uses Matplotlib (`plt`) for plotting.
- Two arrays (`x` and `y`) are defined as NumPy arrays, representing the coordinates of the data points.
- `mycolor` is a list of colors, where each color corresponds to a data point in the scatter plot.
- The scatter plot is created using `plt.scatter()` with `color=mycolor`, which assigns each point in the plot a color from the `mycolor` list based on its position.
- The resulting plot will display points colored in the sequence defined by `mycolor`.

Size

You can change the size of the dots with the s argument.

In [41]:
```
import matplotlib.pyplot as plt
import numpy as np

x = np.array([0,1,2,3,4,5])
y = np.array([0,2,8,1,14,7])
mycolor=['red','green','purple','lime','aqua','yellow']
size=[10,60,120,80,20,190]
plt.scatter(x,y,color= mycolor, s=size )

plt.show()
```


14
12
10
8
6
4
2
0
0
1
2
3
4
5

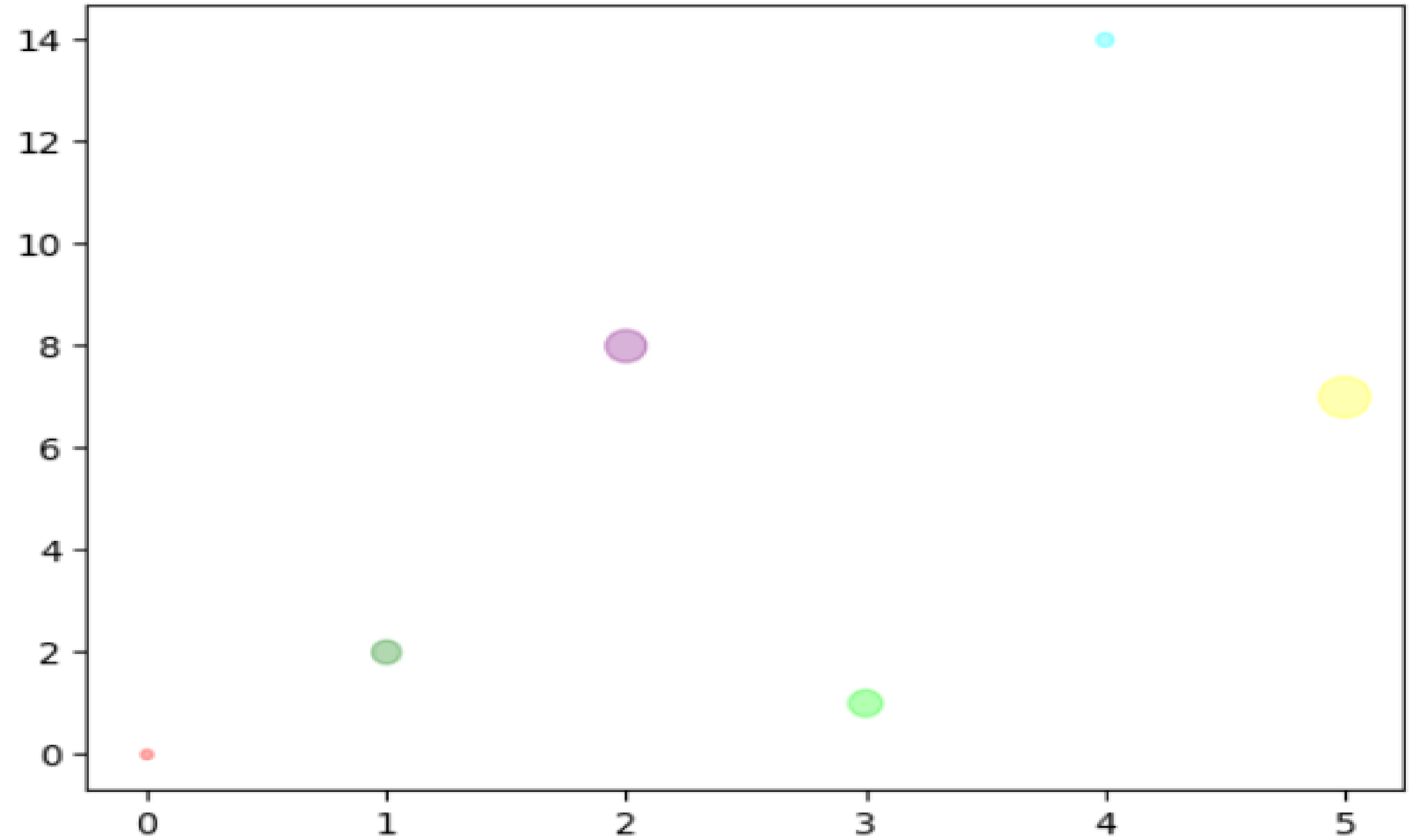
Code Summary:

- The code uses Matplotlib (`plt`) for plotting.
- Two arrays (`x` and `y`) are defined as NumPy arrays, representing the coordinates of the data points.
- `mycolor` is a list of colors, where each color corresponds to a data point in the scatter plot.
- `size` is a list of sizes, where each size value corresponds to a data point in the scatter plot.
- The scatter plot is created using `plt.scatter()` with `color=mycolor` and `s=size`, which assigns each point in the plot a color and size based on the corresponding values in `mycolor` and `size`.
- The resulting plot will display points colored and sized according to the `mycolor` and `size` lists respectively.

Alpha

You can adjust the transparency of the dots with the alpha argument.

In [42]:
```
import matplotlib.pyplot as plt
import numpy as np

x = np.array([0,1,2,3,4,5])
y = np.array([0,2,8,1,14,7])
mycolor=['red','green','purple','lime','aqua','yellow']
size=[10,60,120,80,20,190]
plt.scatter(x,y,color= mycolor, s=size, alpha=0.3 )
plt.show()
```


14
12
10
8
6
4
2
0
0
1
2
3
4
5

Code Summary:

- The code uses Matplotlib (`plt`) for plotting.
- Two arrays (`x` and `y`) are defined as NumPy arrays, representing the coordinates of the data points.
- `mycolor` is a list of colors, where each color corresponds to a data point in the scatter plot.
- `size` is a list of sizes, where each size value corresponds to a data point in the scatter plot.
- The scatter plot is created using `plt.scatter()` with `color=mycolor`, `s=size`, and `alpha=0.3`, which assigns each point in the plot a color, size, and makes them semi-transparent based on the corresponding values in `mycolor`, `size`, and `alpha`.
- The resulting plot will display points colored, sized, and semi-transparent according to the `mycolor`, `size`, and `alpha` parameters respectively.

Create Bar

A bar plot is a type of chart that represents categorical data with rectangular bars where the length or height of the bars corresponds to the value they represent.

Bar plots are useful for comparing quantities of different categories or groups.

Bar plots are widely used in various real-life scenarios to visualize categorical data and compare quantities across different categories. Here are a few real-life examples where bar plots are commonly employed:

1. **Sales Data Analysis**:

    - **Scenario**: A retail store wants to analyze its monthly sales across different product categories (e.g., electronics, clothing, groceries).
    - **Bar Plot Use**: A vertical bar plot can be used to display the total sales figures for each category. This allows stakeholders to quickly compare which product categories are performing better or worse.

2. **Survey Results**:

   - **Scenario**: A company conducts a survey among its customers to gather feedback on product satisfaction (e.g., very satisfied, satisfied, neutral, dissatisfied, very dissatisfied).
   - **Bar Plot Use**: A horizontal bar plot can be used to visualize the distribution of responses across satisfaction levels. This helps in understanding the overall sentiment of customers towards the product or service.

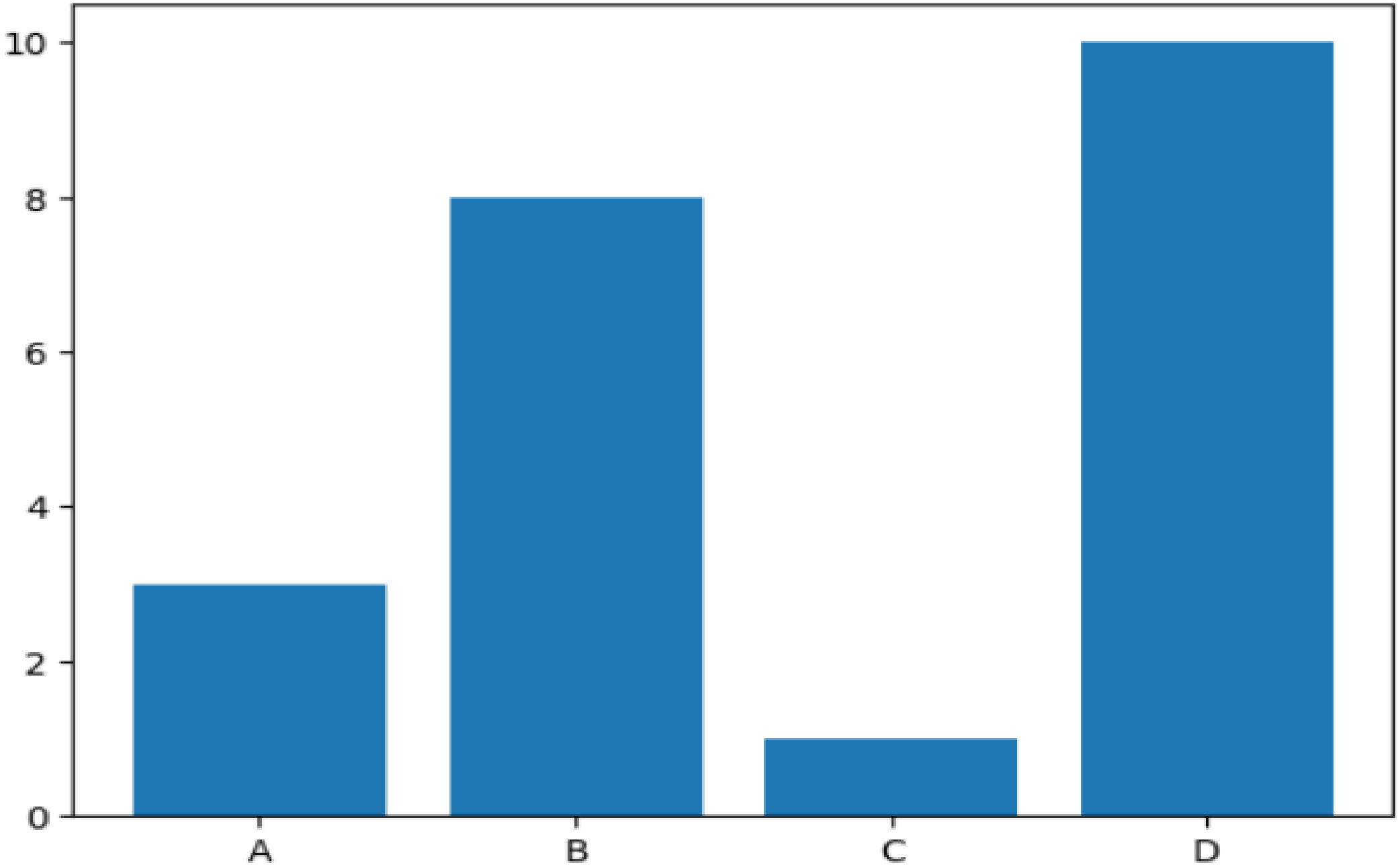
3. **Market Share Analysis**:

   - **Scenario**: A marketing team wants to analyze the market share of different brands within a specific industry segment (e.g., smartphones, automobiles).
   - **Bar Plot Use**: A vertical grouped bar plot can be used to compare the market shares of various brands side by side. This visualization allows for easy identification of market leaders and their relative positions.

Create Bar

With Pyplot, you can use the bar() function to draw bar graphs:

In [44]:
```
import matplotlib.pyplot as plt
import numpy as np


x = np.array(["A","B","C","D"])
y = np.array([3,8,1,10])
plt.bar(x,y)
plt.show()
```


10
8
6
4
2
0
A
B
C
D

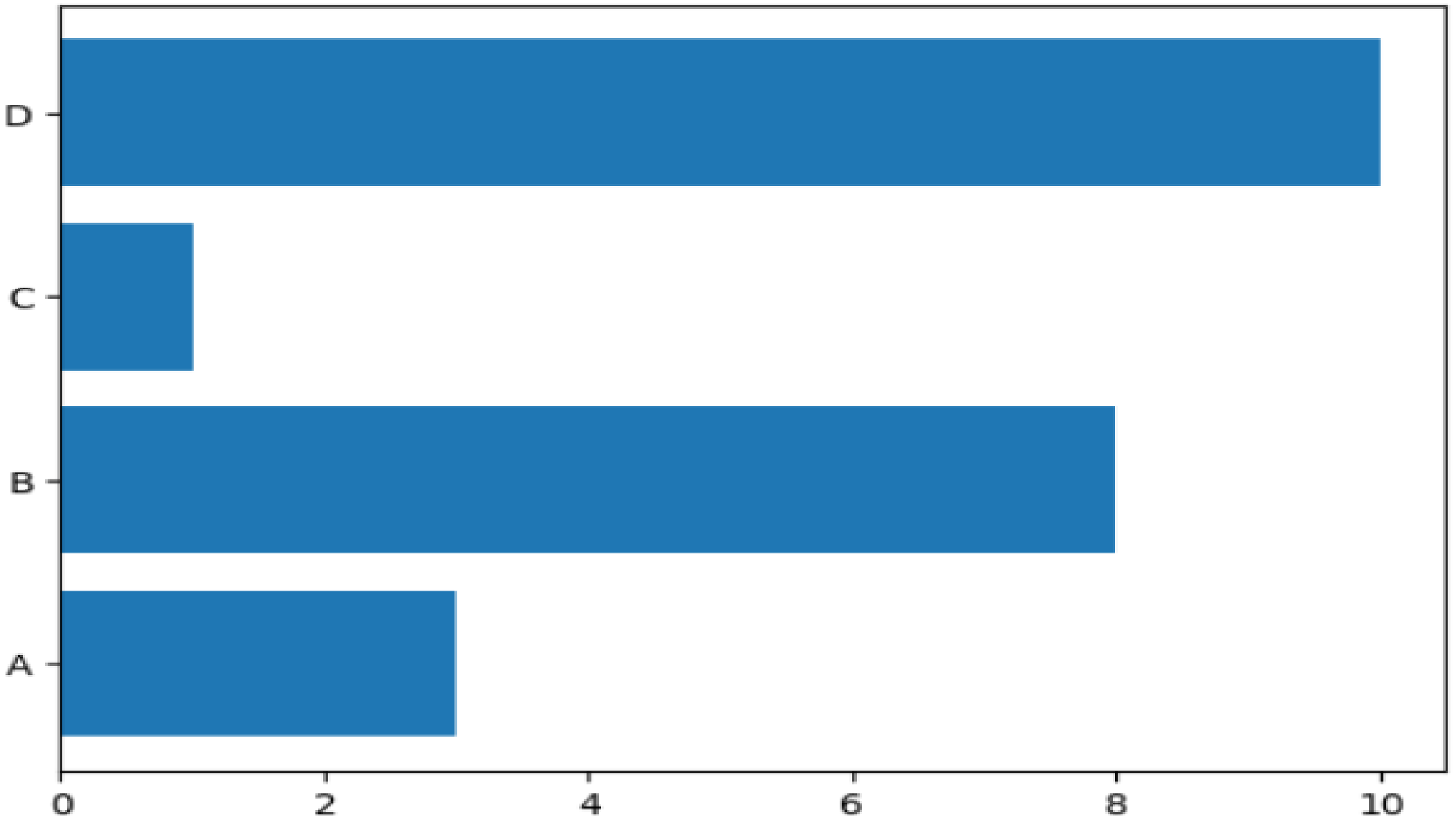
Code Summary:

- The code uses Matplotlib (`plt`) for plotting.
- Two arrays (`x` and `y`) are defined as NumPy arrays, representing the labels/categories and heights/values for the bars in the bar plot.
- The bar plot is created using `plt.bar(x, y)`, where `x` is the array of labels/categories and `y` is the array of heights/values.
- The resulting plot will display a bar chart with bars labeled `"A"`, `"B"`, `"C"`, `"D"` and heights corresponding to `[3, 8, 1, 10]`.

Create Horizontal Bar

If you want the bars to be displayed horizontally instead of vertically, use the barh() function:

In [46]:
```
import matplotlib.pyplot as plt
import numpy as np


x = np.array(["A","B","C","D"])
y = np.array([3,8,1,10])
plt.barh(x,y)
plt.show()
```


D
C
B
A
0
2
4
6
8
10

Code Summary:

- The code uses Matplotlib (`plt`) for plotting.
- Two arrays (`x` and `y`) are defined as NumPy arrays, representing the labels/categories and widths/values for the horizontal bars in the plot.
- The horizontal bar plot is created using `plt.barh(x, y)`, where `x` is the array of labels/categories and `y` is the array of widths/values.
- The resulting plot will display a horizontal bar chart with bars labeled `"A"`, `"B"`, `"C"`, `"D"` and widths corresponding to `[3, 8, 1, 10]`.

Histogram

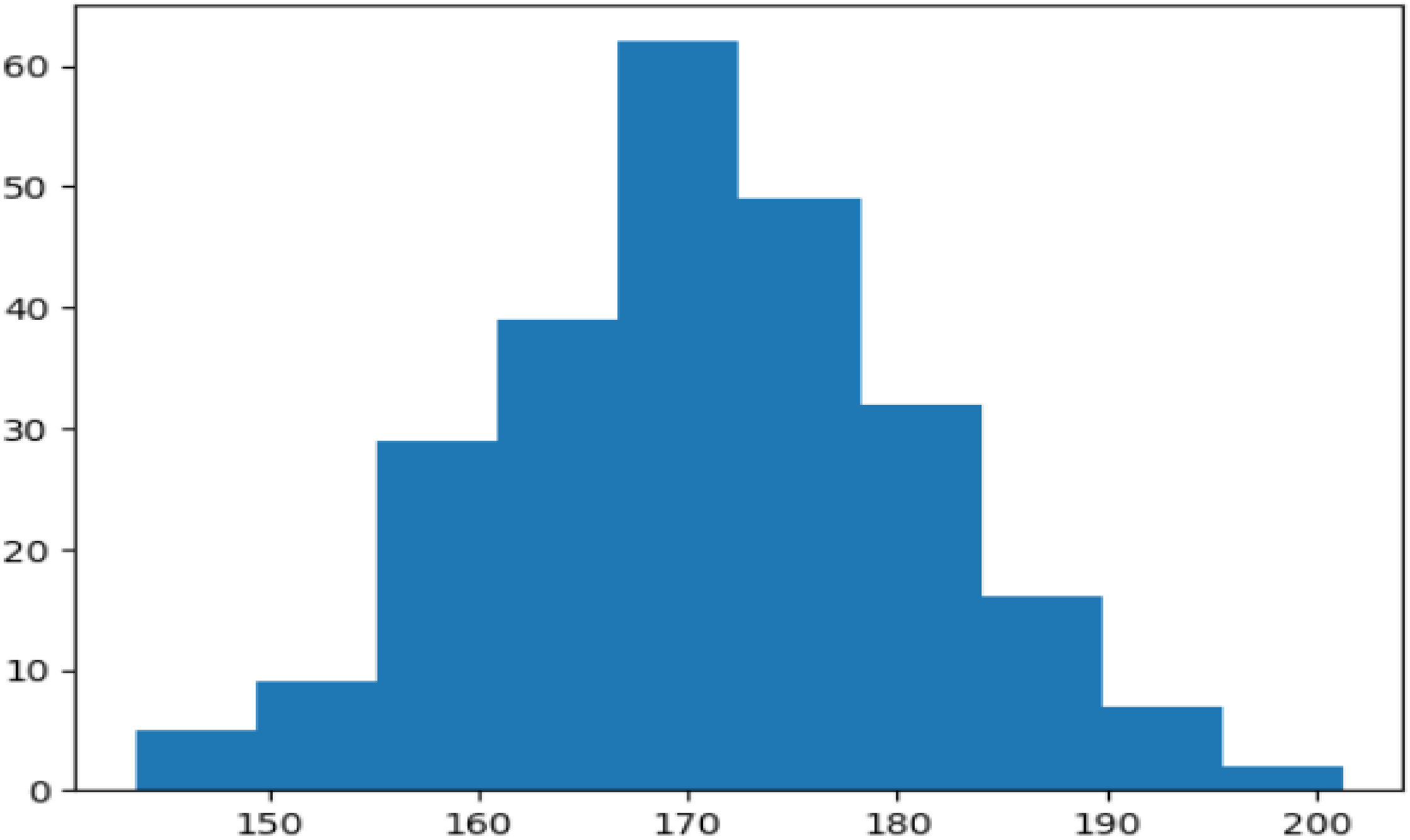
- A histogram is a graph showing frequency distributions.
- It is a graph showing the number of observations within each given interval.
- In Matplotlib, we use the hist() function to create histograms.
- The hist() function will read the array and produce a histogram:

Characteristics of a Histogram:

- **Bins**: The data range is divided into intervals called bins or buckets.
- **Bar Heights**: The height of each bar represents the frequency or count of data points within that bin.
- **Continuous Data**: Histograms are used for continuous data where the values are numerical and ordered.
- **No Gaps**: There are no gaps between the bars, as opposed to bar charts which typically represent categorical data.

Histogram

In [50]:
```
import matplotlib.pyplot as plt
import numpy as np

x = np.random.normal(170,10,250)
plt.hist(x)
plt.show()
```


60
50
40
30
20
10
0
150
160
170
180
190
200

Explanation:

1. Importing Libraries:

   - `import matplotlib.pyplot as plt`: Imports the Matplotlib library, specifically the `pyplot` module, and aliases it as `plt` for easier usage.
   - `import numpy as np`: Imports the NumPy library and aliases it as `np`, which is commonly used for numerical operations and array handling.

2. Generating Random Data:

   - `x = np.random.normal(170, 10, 250)`: Generates 250 random numbers from a normal (Gaussian) distribution with a mean (center) of 170 and a standard deviation (spread) of 10. This simulates a dataset where values are clustered around 170 with some variation.

3. **Creating the Histogram:**

   - `plt.hist(x)`: Creates a histogram of the data stored in `x`. By default, Matplotlib will automatically determine the number of bins (bars) based on the data and display the histogram.

4. **Displaying the Plot:**

   - `plt.show()`: Displays the histogram plot on the screen. This function is necessary to render any Matplotlib plot interactively.

In a normal distribution, the standard deviation plays a crucial role in defining the spread or dispersion of the data around the mean. Here's a concise rule that is commonly used to interpret the spread of data in a normal distribution based on the standard deviation:

1. **68-95-99.7 Rule (Empirical Rule):**
    - Approximately 68% of the data falls within one standard deviation (σ) of the mean (μ).
    - Approximately 95% of the data falls within two standard deviations (2σ) of the mean (μ).
    - Approximately 99.7% of the data falls within three standard deviations (3σ) of the mean (μ).

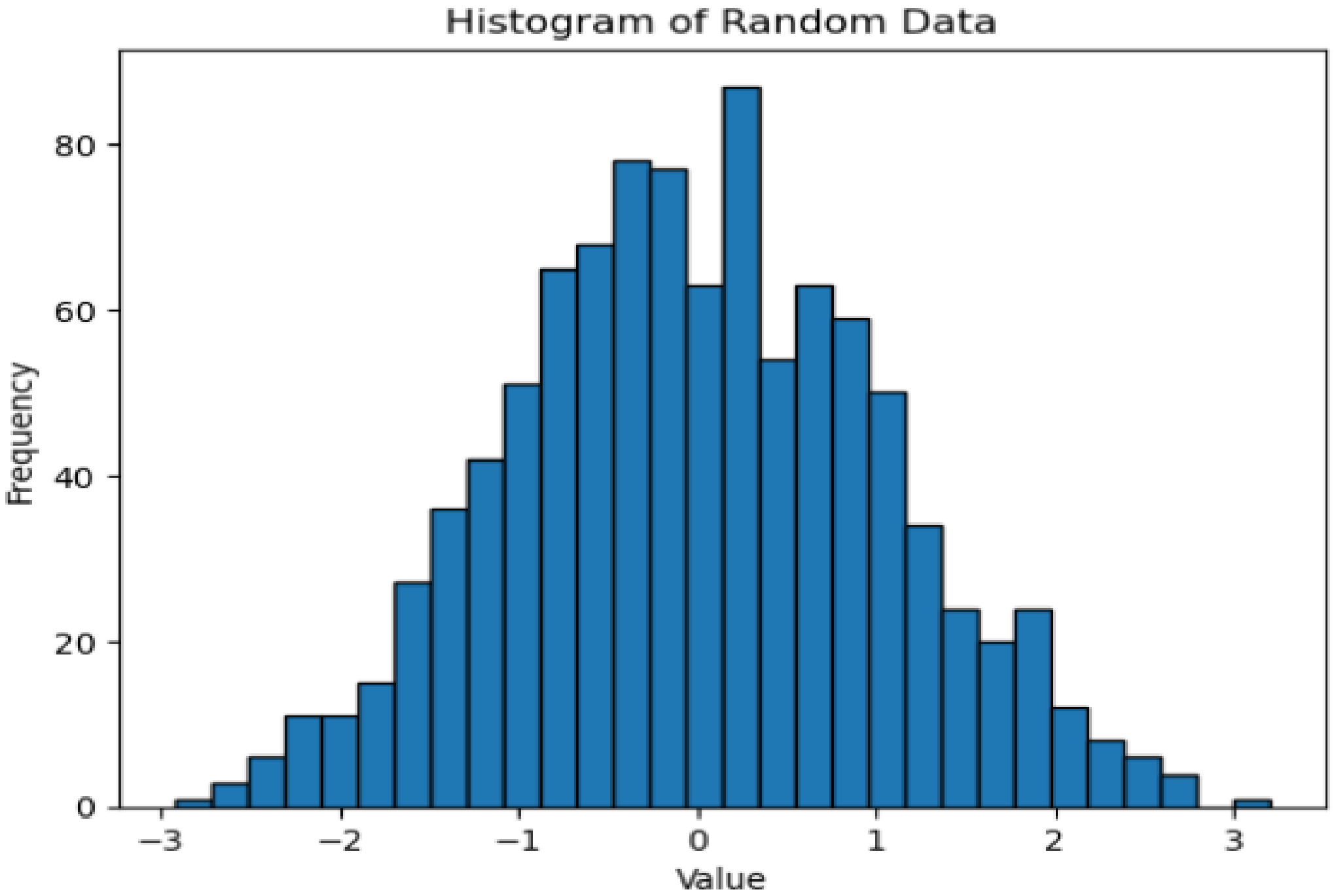
Interpretation:

- **68% Rule:** If a normal distribution has a mean of μ and a standard deviation of σ, roughly 68% of the data points will fall within the interval $[\mu - \sigma, \mu + \sigma]$.

- **95% Rule:** About 95% of the data points will fall within the interval $[\mu - 2\sigma, \mu + 2\sigma]$.

- **99.7% Rule:** Nearly all (99.7%) of the data points will fall within the interval $[\mu - 3\sigma, \mu + 3\sigma]$.

[1]:
```python
import matplotlib.pyplot as plt
import numpy as np

# Generate random data (normal distribution)
data = np.random.normal(loc=0, scale=1, size=1000)

# Create histogram
plt.hist(data, bins=30, edgecolor='black')  # Using 30 bins for demonstration

# Add labels and title
plt.xlabel('Value')
plt.ylabel('Frequency')
plt.title('Histogram of Random Data')

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



1. Imports:

   - `import matplotlib.pyplot as plt`: Imports Matplotlib's pyplot module for creating plots.
   - `import numpy as np`: Imports NumPy for generating random data.

2. Generate Random Data:

   - `data = np.random.normal(loc=0, scale=1, size=1000)`: Generates 1000 random numbers from a normal distribution (`np.random.normal`).
     - `loc=0`: Specifies the mean of the distribution (centered at 0).
     - `scale=1`: Specifies the standard deviation of the distribution (spread of 1).
     - `size=1000`: Specifies the number of random numbers to generate (1000 in this case).

3. Create Histogram:

- `plt.hist(data, bins=30, edgecolor='black')`: Creates a histogram using the generated `data`.
  - `data`: The array of data points to be plotted as a histogram.
  - `bins=30`: Specifies the number of bins (intervals) to use in the histogram. More bins give more detail but can make the plot more noisy.
  - `edgecolor='black'`: Sets the color of the edges of the bars in the histogram for better visibility.

4. **Customize Labels and Title:**

   - `plt.xlabel('Value')`: Adds a label to the x-axis, indicating the values being plotted (here, the random data points).
   - `plt.ylabel('Frequency')`: Adds a label to the y-axis, indicating the frequency or count of data points in each bin.
   - `plt.title('Histogram of Random Data')`: Adds a title to the plot, describing what the plot shows.

5. **Display the Plot:**

   - `plt.show()`: Displays the histogram plot in the Python environment.

Creating Pie Charts

With Pyplot, you can use the pie() function to draw pie charts:

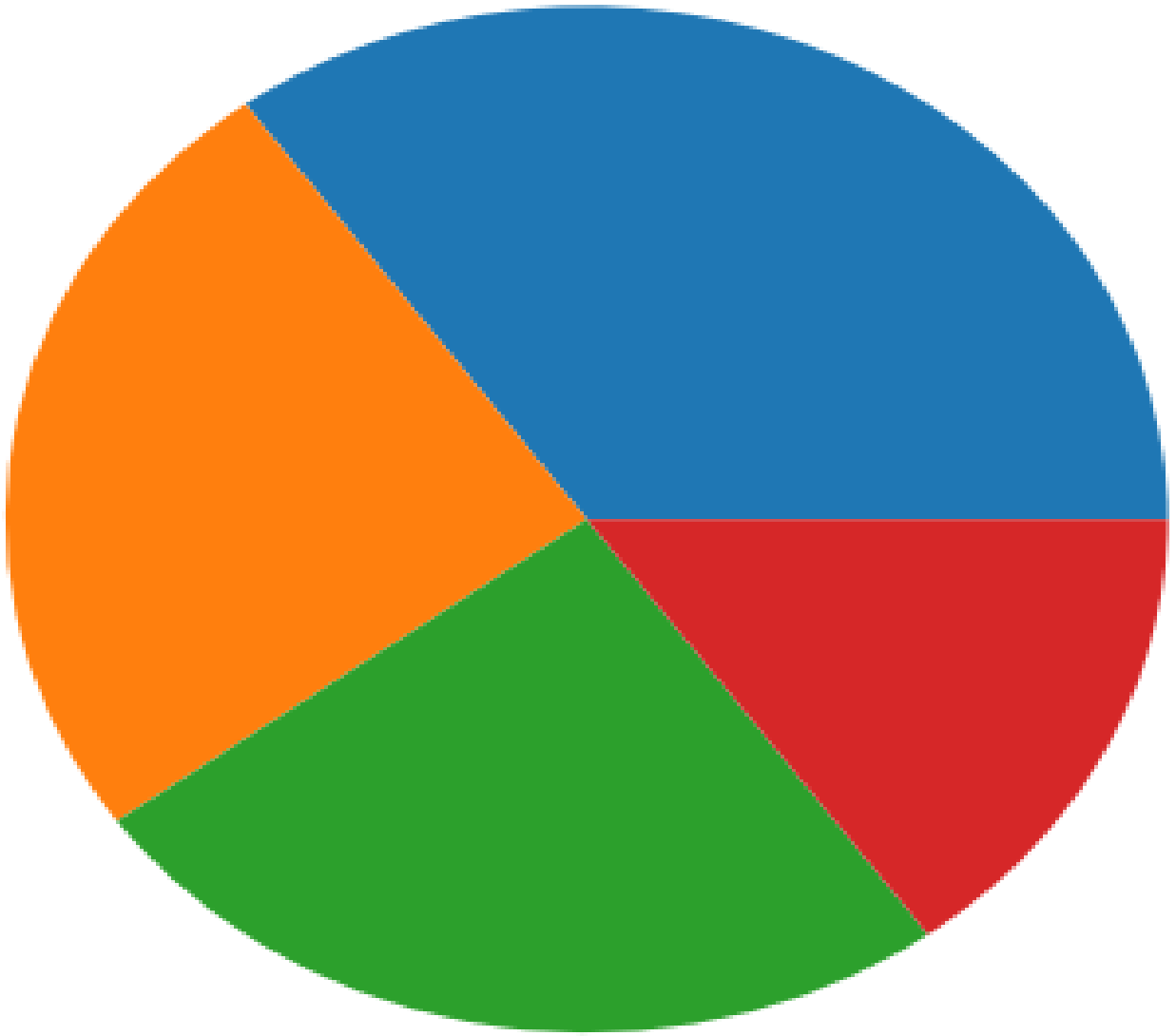
Characteristics of a Pie Chart:

- **Circular Representation**: The entire pie represents 100% of the data.
- **Slices**: Each slice represents a category or group, and its size is proportional to the percentage or fraction it represents.
- **Labels**: Labels often accompany each slice to indicate the category name and its percentage or value.
- **Sum to 100%**: The sum of all slices' percentages equals 100%.

Creating Pie Charts

With Pyplot, you can use the pie() function to draw pie charts:

[34]:
```
import matplotlib.pyplot as plt
import numpy as np


x=np.array([35,25,25,15])
plt.pie(x)
plt.show()
```



Explanation:

1. Importing Libraries:

   - `import matplotlib.pyplot as plt`: Imports the Matplotlib library, specifically the `pyplot` module, and aliases it as `plt` for easier usage.
   - `import numpy as np`: Imports the NumPy library and aliases it as `np`, which is commonly used for numerical operations and array handling.

2. Data Array:

   - `x = np.array([35, 25, 25, 15])`: Defines an array `x` containing four numerical values `[35, 25, 25, 15]`. These values represent the sizes or proportions of different categories or slices in the pie chart.

3. **Creating the Pie Chart:**

- `plt.pie(x)`: Generates a pie chart using the data from `x`. Each value in `x` corresponds to the size (in percentage) of a slice in the pie chart. The slices are automatically labeled and colored by Matplotlib.

4. **Displaying the Plot:**

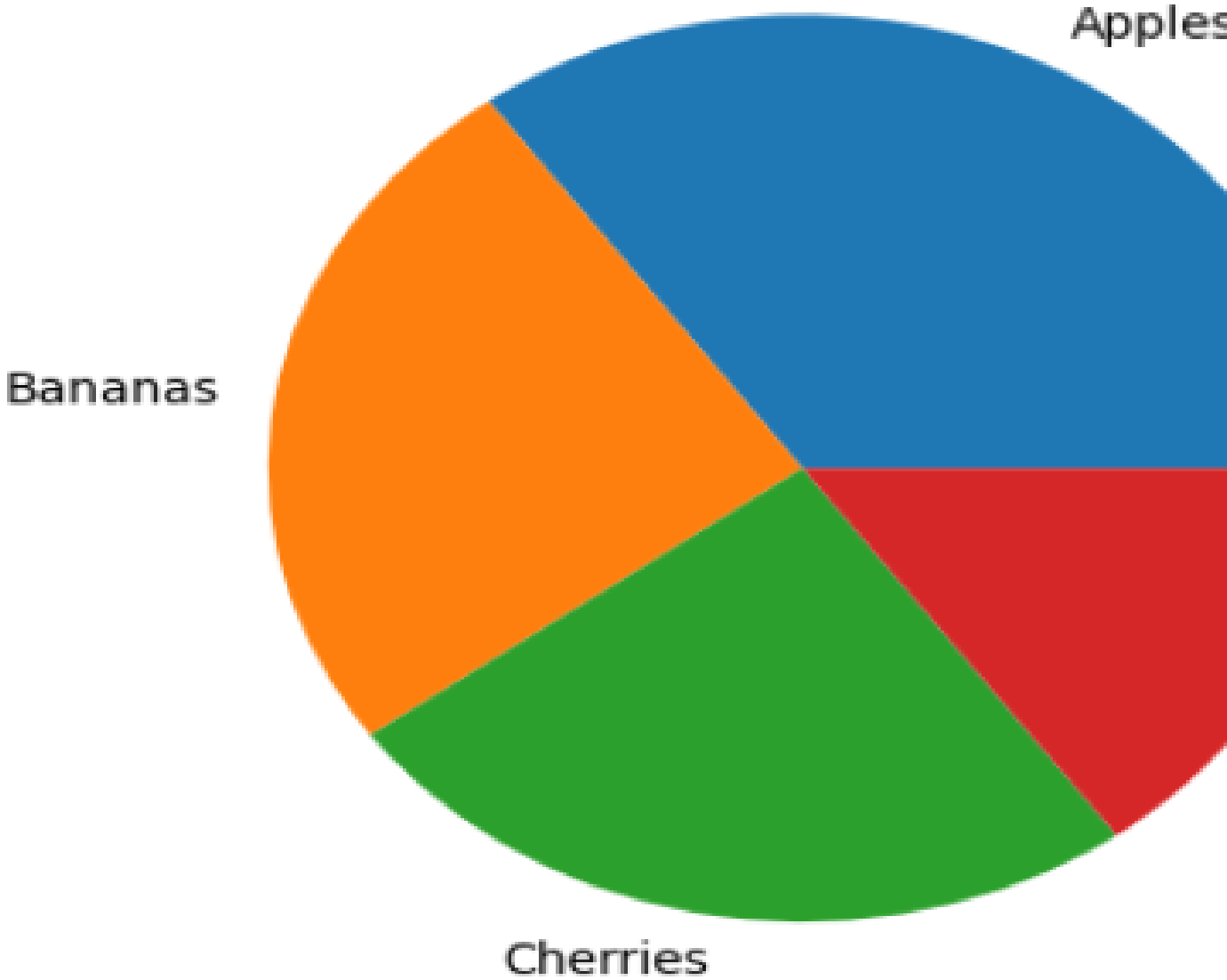
- `plt.show()`: Displays the pie chart on the screen. This function is necessary to render any Matplotlib plot interactively.

Labels

Add labels to the pie chart with the label parameter.

The label parameter must be an array with one label for each wedge:

In [5]:
```
import matplotlib.pyplot as plt
import numpy as np

x = np.array([35,25,25,15])
mylabels = ["Apples", "Bananas", "Cherries", "Dates"]
plt.pie(x, labels=mylabels)
plt.show()
```

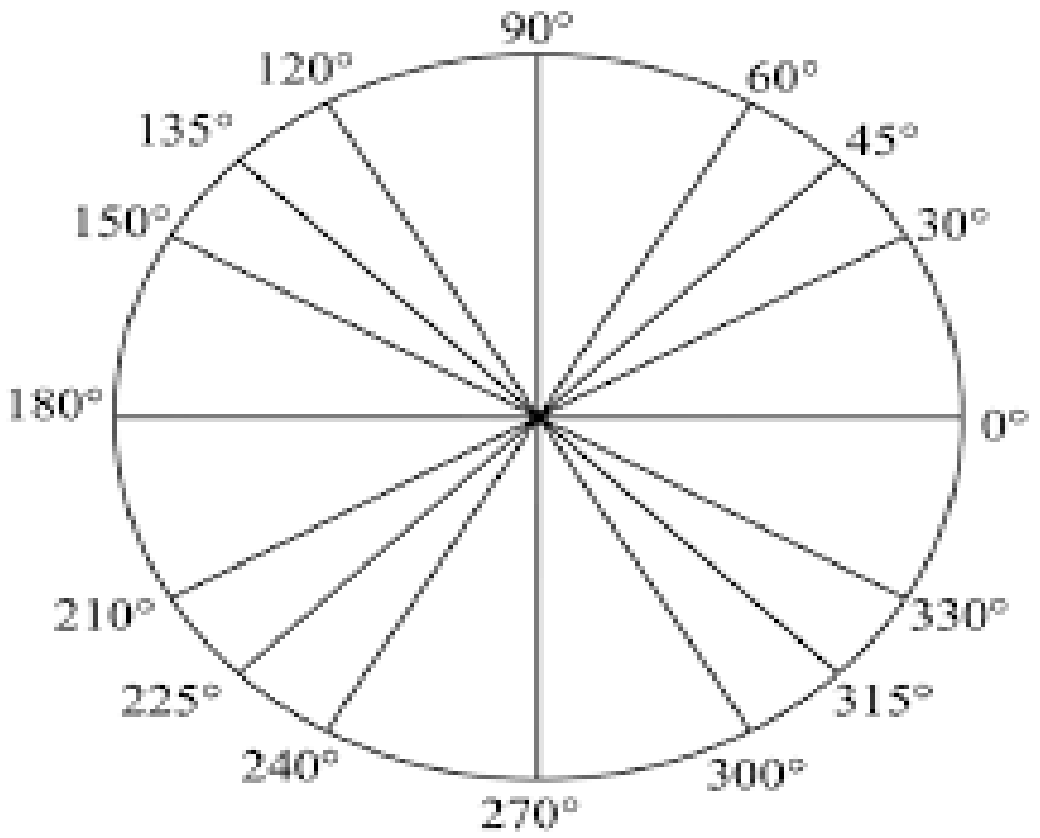

Apples
Bananas
Dates
Cherries

Default Behavior of `plt.pie(x)`

1. **Starting Angle:**
    - The first slice (corresponding to the first value in `x`) starts at the 0-degree angle, which is the 3 o'clock position on the circle.
2. **Slicing Order:**
    - Slices are plotted in the order of the elements in `x`. For example, if `x = [35, 25, 25, 15]`, then the slices are drawn in the order corresponding to these values.
3. **Automatic Colors and Legend:**
    - Matplotlib automatically assigns different colors to each slice and includes a legend by default that labels each slice with its corresponding value from `x`.

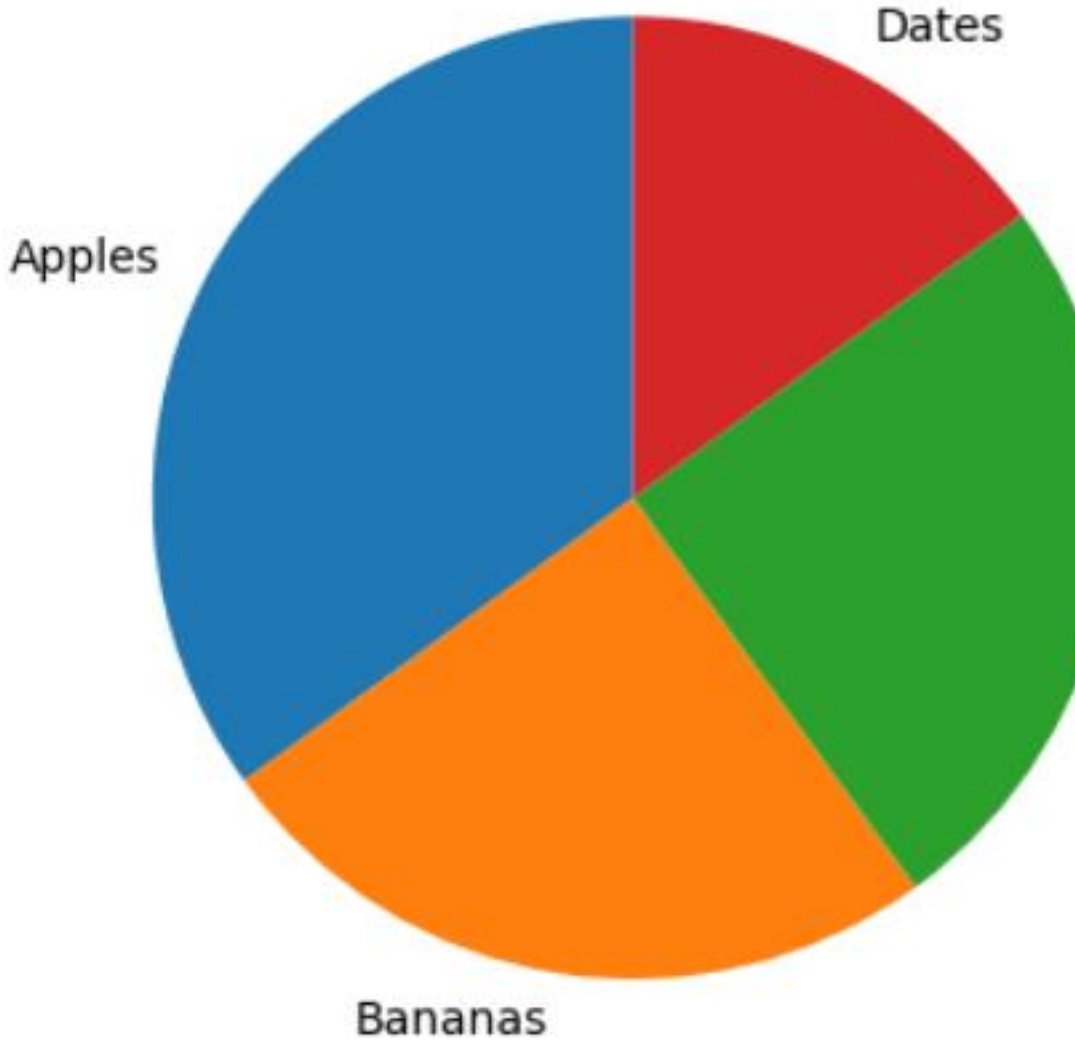
Start Angle

As mentioned the default start angle is at the x-axis, but you can change the start angle by specifying a startangle parameter.



[30]:

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


x=np.array([35,25,25,15])
mylabels=["Apples","Bananas","Cherries","Dates"]
plt.pie(x,labels=mylabels,startangle=90)
plt.show()
```


Dates
Apples
Cherries
Bananas

1. **Importing Libraries**:

   - `` `import matplotlib.pyplot as plt` ``: Imports the Matplotlib library, specifically the `` `pyplot` `` module, and aliases it as `` `plt` `` for easier usage.
   - `` `import numpy as np` ``: Imports the NumPy library and aliases it as `` `np` ``, which is commonly used for numerical operations and array handling.

2. **Data Array and Labels**:

   - `` `x = np.array([35, 25, 25, 15])` ``: Defines an array `` `x` `` containing four numerical values `` `[35, 25, 25, 15]` ``. These values represent the sizes or proportions of different categories or slices in the pie chart.
   - `` `mylabels = ["Apples", "Bananas", "Cherries", "Dates"]` ``: Defines `` `mylabels` `` as a list of strings, providing custom labels for each corresponding slice in the pie chart.

3. **Creating the Pie Chart:**

   - `plt.pie(x, labels=mylabels, startangle=90)`: Generates a pie chart using the data from `x` and labels each slice with the corresponding string from `mylabels`. The `startangle=90` parameter specifies the angle at which the first slice starts. By default, this angle is 0 degrees, starting from the positive x-axis and moving counterclockwise. Setting `startangle=90` rotates the starting position by 90 degrees counterclockwise, making the first slice vertically oriented.

4. **Displaying the Plot:**

   - `plt.show()`: Displays the pie chart on the screen. This function is necessary to render any Matplotlib plot interactively.

Output Interpretation:

When you run this code, you will see a pie chart where:

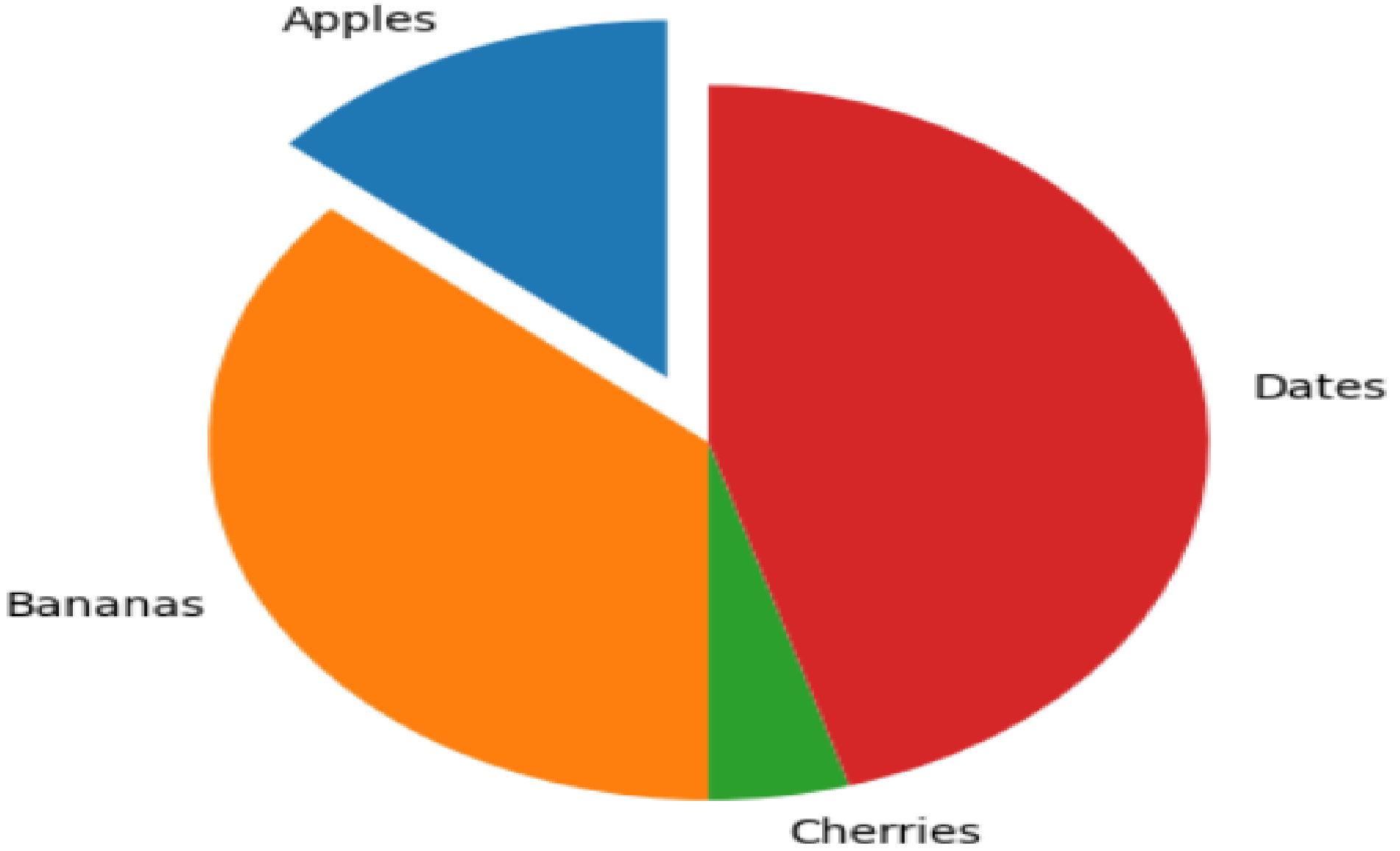
- Each slice represents a proportion of the whole pie (100%).
- The proportions are based on the values in the array `x`.
- The labels (`mylabels`) are displayed next to each slice.
- The chart starts with the "Apples" slice at the 90-degree angle counterclockwise from the positive x-axis.

Explode

Maybe you want one of the wedges to stand out? The explode parameter allows you to do that.

The explode parameter, if specified, and not None, must be an array with one value for each wedge

In [18]:
```
import matplotlib.pyplot as plt
import numpy as np
x = np.array([35,25,25,15])
mylabels = ["Apples", "Bananas", "Cherries", "Dates"]
myexplode = [0.2, 0, 0, 0]
plt.pie(y, labels=mylabels, startangle=90,explode=myexplode)
plt.show()
```


Apples
Dates
Bananas
Cherries

Explanation:

1. Importing Libraries:

   - `import matplotlib.pyplot as plt`: Imports the Matplotlib library, specifically the `pyplot` module, and aliases it as `plt` for easier usage.
   - `import numpy as np`: Imports the NumPy library and aliases it as `np`, which is commonly used for numerical operations and array handling.

2. Data Array, Labels, and Explode Parameter:

- `x = np.array([35, 25, 25, 15])`: Defines an array `x` containing four numerical values `[35, 25, 25, 15]`. These values represent the sizes or proportions of different categories or slices in the pie chart.
- `mylabels = ["Apples", "Bananas", "Cherries", "Dates"]`: Defines `mylabels` as a list of strings, providing custom labels for each corresponding slice in the pie chart.
- `myexplode = [0.2, 0, 0, 0]`: Defines `myexplode` as a list of float values representing the fraction of the radius with which to offset each wedge. In this case, `[0.2, 0, 0, 0]` means that only the first slice ("Apples") will be exploded (pulled out from the pie chart).

3. **Creating the Pie Chart:**

   - `plt.pie(x, labels=mylabels, startangle=90, explode=myexplode)`: Generates a pie chart using the data from `x`, labels each slice with the corresponding string from `mylabels`, sets the starting angle at 90 degrees (rotated counterclockwise from the positive x-axis), and explodes the slices as specified in `myexplode`.

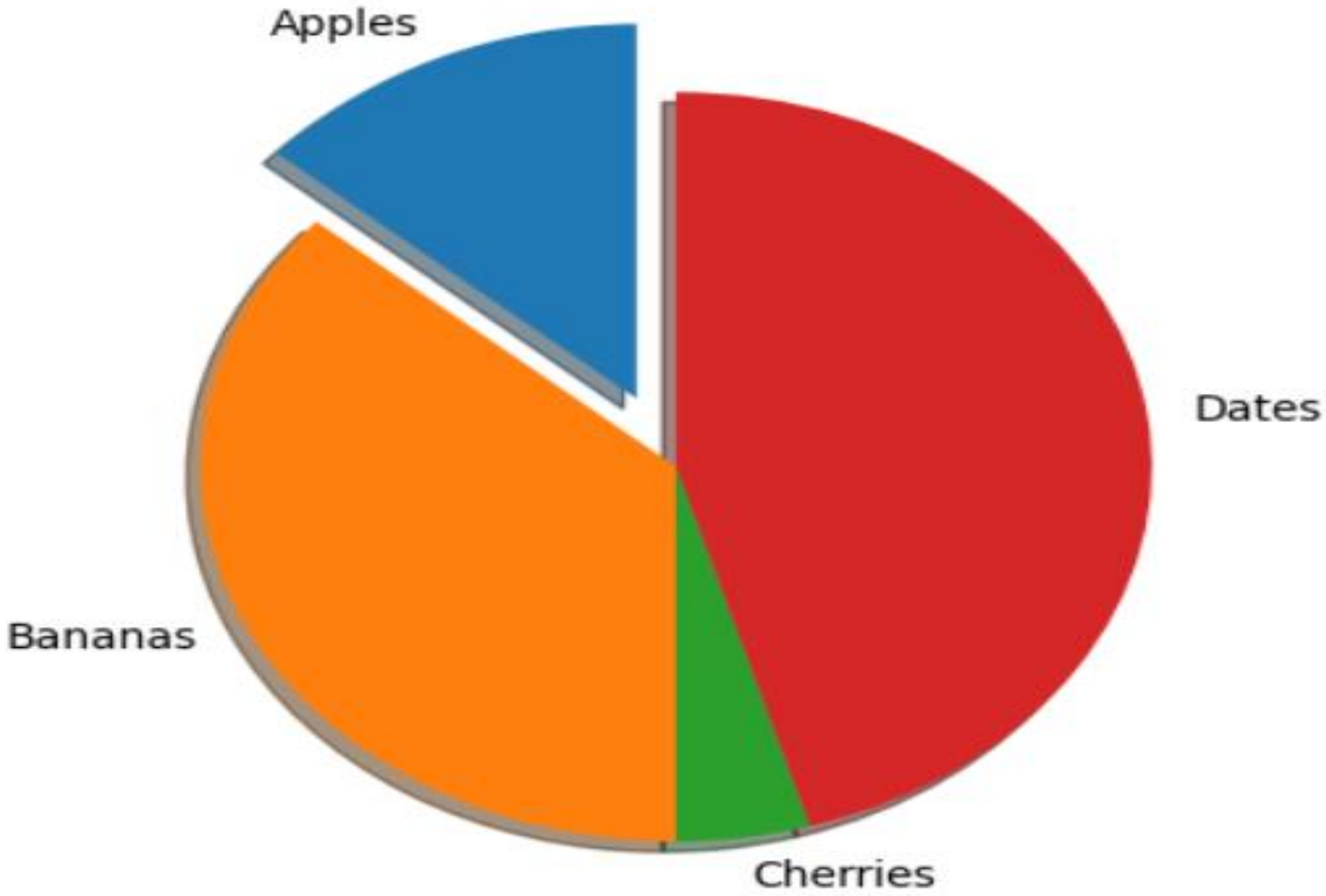
4. **Displaying the Plot:**

   - `plt.show()`: Displays the pie chart on the screen. This function is necessary to render any Matplotlib plot interactively.

Shadow

Add a shadow to the pie chart by setting the shadows parameter to True:

In [14]:
```
import matplotlib.pyplot as plt
import numpy as np
x = np.array([35,25,25,15])
mylabels = ["Apples", "Bananas", "Cherries", "Dates"]
myexplode = [0.2, 0, 0, 0]
plt.pie(y, labels=mylabels, startangle=90, explode=myexplode, shadow=True)
plt.show()
```



Explanation:

1. Importing Libraries:

   - `import matplotlib.pyplot as plt`: Imports the Matplotlib library, specifically the `pyplot` module, and aliases it as `plt` for easier usage.
   - `import numpy as np`: Imports the NumPy library and aliases it as `np`, which is commonly used for numerical operations and array handling.

2. **Data Array, Labels, Explode Parameter, and Shadow:**

- `x = np.array([35, 25, 25, 15])`: Defines an array `x` containing four numerical values `[35, 25, 25, 15]`. These values represent the sizes or proportions of different categories or slices in the pie chart.
- `mylabels = ["Apples", "Bananas", "Cherries", "Dates"]`: Defines `mylabels` as a list of strings, providing custom labels for each corresponding slice in the pie chart.
- `myexplode = [0.2, 0, 0, 0]`: Defines `myexplode` as a list of float values representing the fraction of the radius with which to offset each wedge. In this case, `[0.2, 0, 0, 0]` means that only the first slice ("Apples") will be exploded (pulled out from the pie chart).
- `shadow=True`: Adds a shadow effect to the pie chart for a 3D appearance, making it visually appealing and easier to distinguish slices.

3. **Creating the Pie Chart:**

   - `plt.pie(x, labels=mylabels, startangle=90, explode=myexplode, shadow=True)`: Generates a pie chart using the data from `x`, labels each slice with the corresponding string from `mylabels`, sets the starting angle at 90 degrees (rotated counterclockwise from the positive x-axis), explodes the slices as specified in `myexplode`, and adds a shadow effect to enhance visibility.

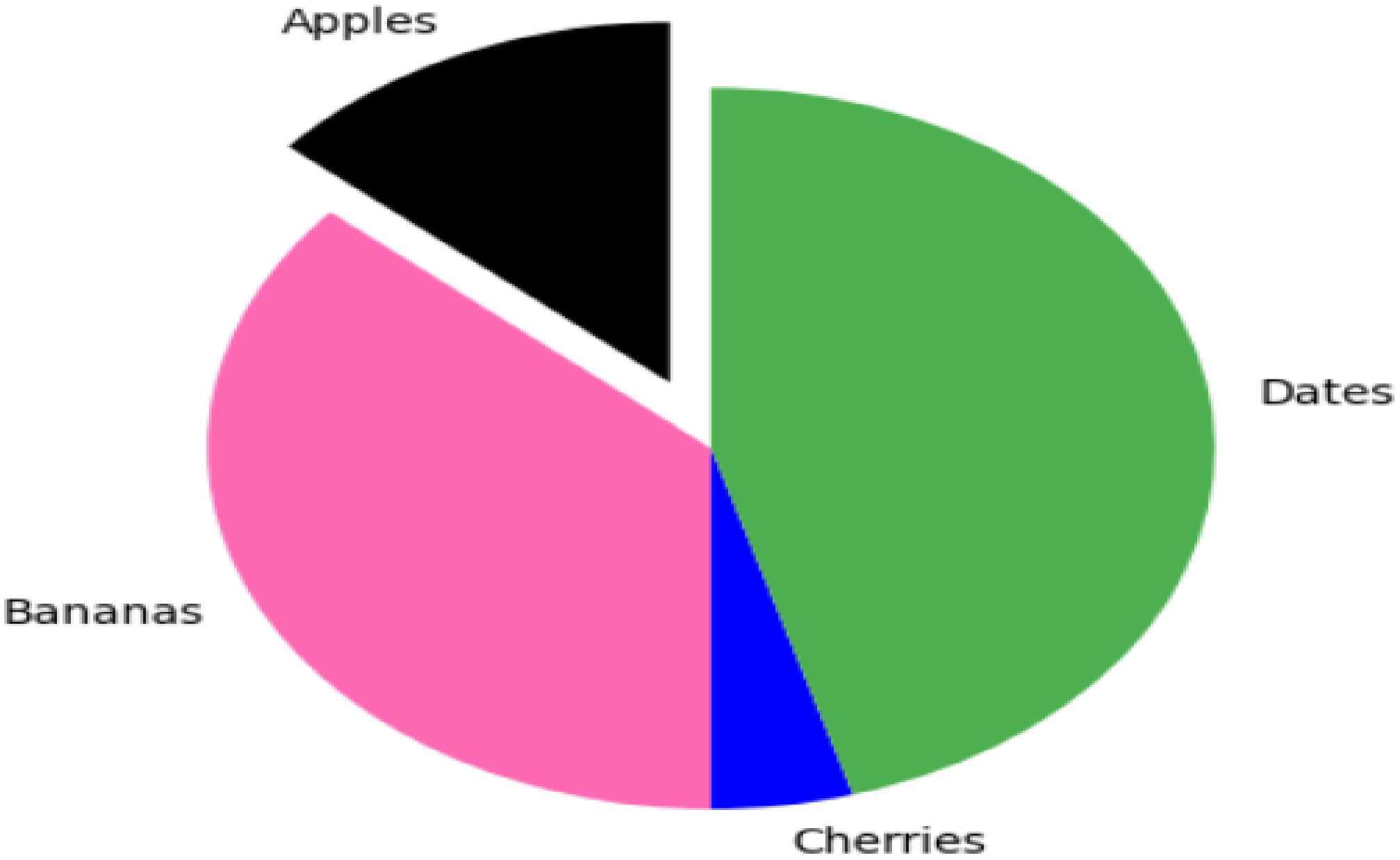
4. **Displaying the Plot:**

   - `plt.show()`: Displays the pie chart on the screen. This function is necessary to render any Matplotlib plot interactively.

Colors

You can set the color of each wedge with the colors parameter.

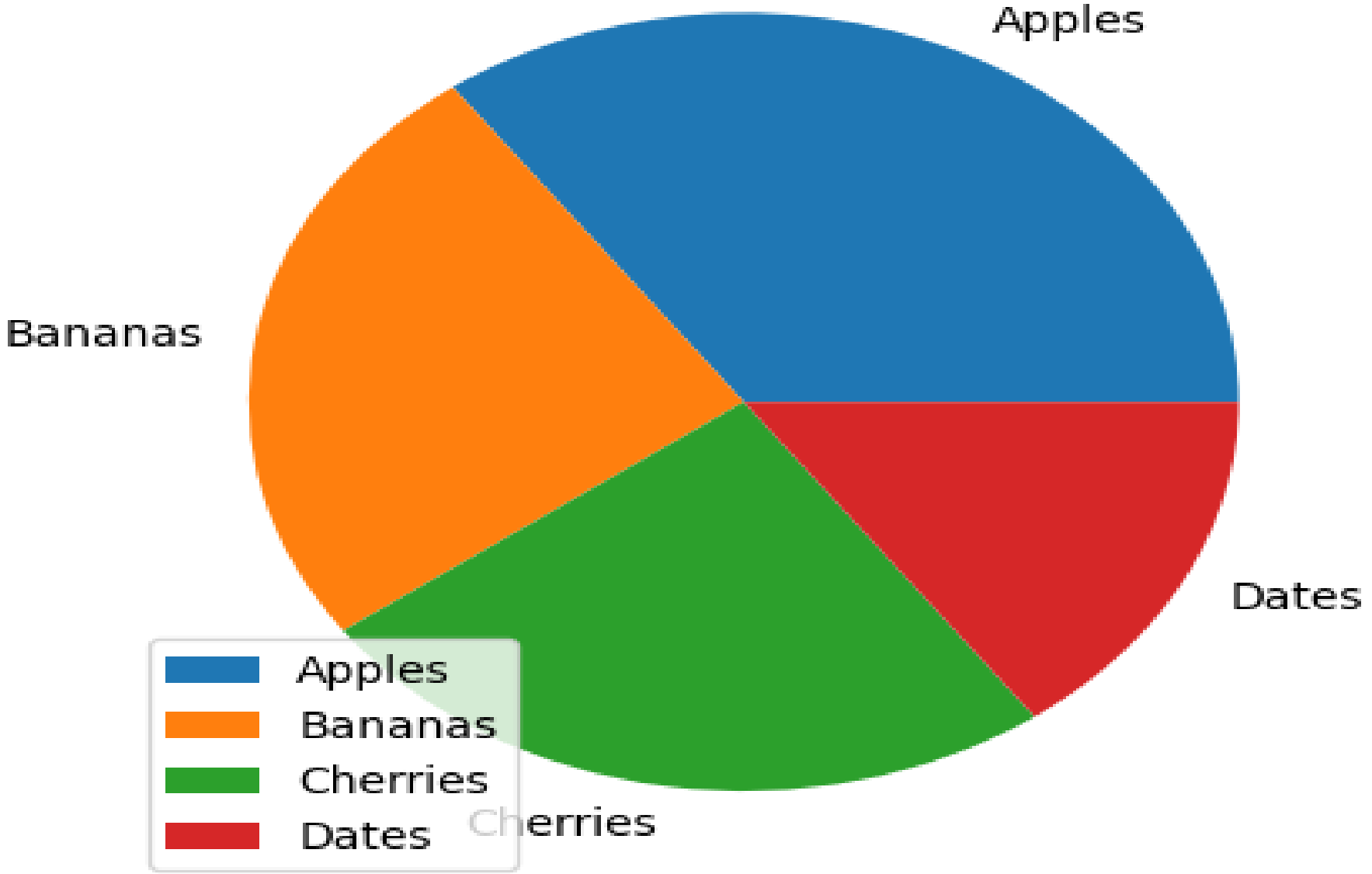
In [15]:
```
import matplotlib.pyplot as plt
import numpy as np
x = np.array([35,25,25,15])
mylabels = ["Apples", "Bananas", "Cherries", "Dates"]
mycolors = ["black", "hotpink", "b", "#4CAF50"]
myexplode = [0.2, 0, 0, 0]
plt.pie(y, labels=mylabels, startangle=90, explode=myexplode, colors= mycolors)
plt.show()
```


Apples
Dates
Bananas
Cherries

Legend

To add a list of explanation for each wedge, use the legend() function:

[46]:
```
import matplotlib.pyplot as plt
import numpy as np

x=np.array([35,25,25,15])
mylabels=["Apples","Bananas","Cherries","Dates"]
plt.pie(x,labels=mylabels)
plt.legend()
plt.show()
```


Apples
Bananas
Cherries
Dates
Apples
Bananas
Cherries
Dates

Explanation:

1. Importing Libraries:

   - `import matplotlib.pyplot as plt`: Imports the Matplotlib library, specifically the `pyplot` module, and aliases it as `plt` for easier usage.
   - `import numpy as np`: Imports the NumPy library and aliases it as `np`, which is commonly used for numerical operations and array handling.

2. **Data Array and Labels:**

   - `` `x = np.array([35, 25, 25, 15])` ``: Defines an array `x` containing four numerical values `[35, 25, 25, 15]`. These values represent the sizes or proportions of different categories or slices in the pie chart.
   - `` `mylabels = ["Apples", "Bananas", "Cherries", "Dates"]` ``: Defines `mylabels` as a list of strings, providing custom labels for each corresponding slice in the pie chart.

3. **Creating the Pie Chart:**

   - `` `plt.pie(x, labels=mylabels)` ``: Generates a pie chart using the data from `x` and labels each slice with the corresponding string from `mylabels`. Matplotlib automatically assigns colors to each slice.

4. Adding a Legend:

- `plt.legend()`: Adds a legend to the plot. The legend explains the colors used for each slice in the pie chart based on the labels provided in `mylabels`.

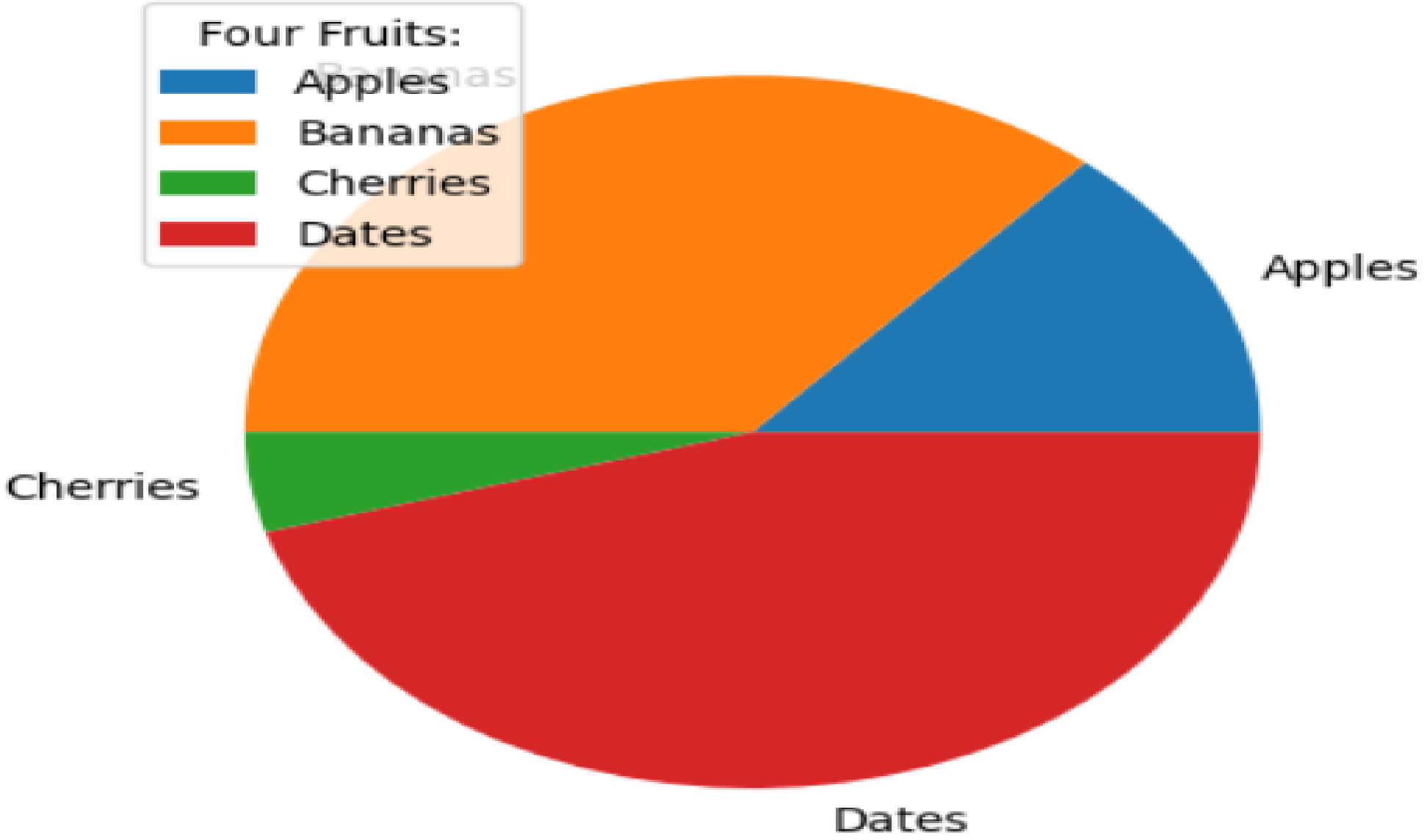
5. Displaying the Plot:

- `plt.show()`: Displays the pie chart on the screen. This function is necessary to render any Matplotlib plot interactively.

Legend With Header

To add a header to the legend, add the title parameter to the legend function.

In [17]:
```
import matplotlib.pyplot as plt
import numpy as np
x = np.array([35,25,25,15])
mylabels = ["Apples", "Bananas", "Cherries", "Dates"]
plt.pie(y, labels=mylabels)
plt.legend(title="Four Fruits:")
plt.show()
```


Four Fruits:
Apples
Bananas
Cherries
Dates
Apples
Cherries
Dates

Explanation:

1. **Importing Libraries:**

   - `` `import matplotlib.pyplot as plt` ``: Imports the Matplotlib library, specifically the `` `pyplot` `` module, and aliases it as `` `plt` `` for easier usage.
   - `` `import numpy as np` ``: Imports the NumPy library and aliases it as `` `np` ``, which is commonly used for numerical operations and array handling.

2. **Data Array and Labels:**

   - `` `x = np.array([35, 25, 25, 15])` ``: Defines an array `` `x` `` containing four numerical values `` `[35, 25, 25, 15]` ``. These values represent the sizes or proportions of different categories or slices in the pie chart.
   - `` `mylabels = ["Apples", "Bananas", "Cherries", "Dates"]` ``: Defines `` `mylabels` `` as a list of strings, providing custom labels for each corresponding slice in the pie chart.

3. **Creating the Pie Chart:**

   - `plt.pie(x, labels=mylabels)`: Generates a pie chart using the data from `x` and labels each slice with the corresponding string from `mylabels`. Matplotlib automatically assigns colors to each slice.

4. **Adding a Legend with Title:**

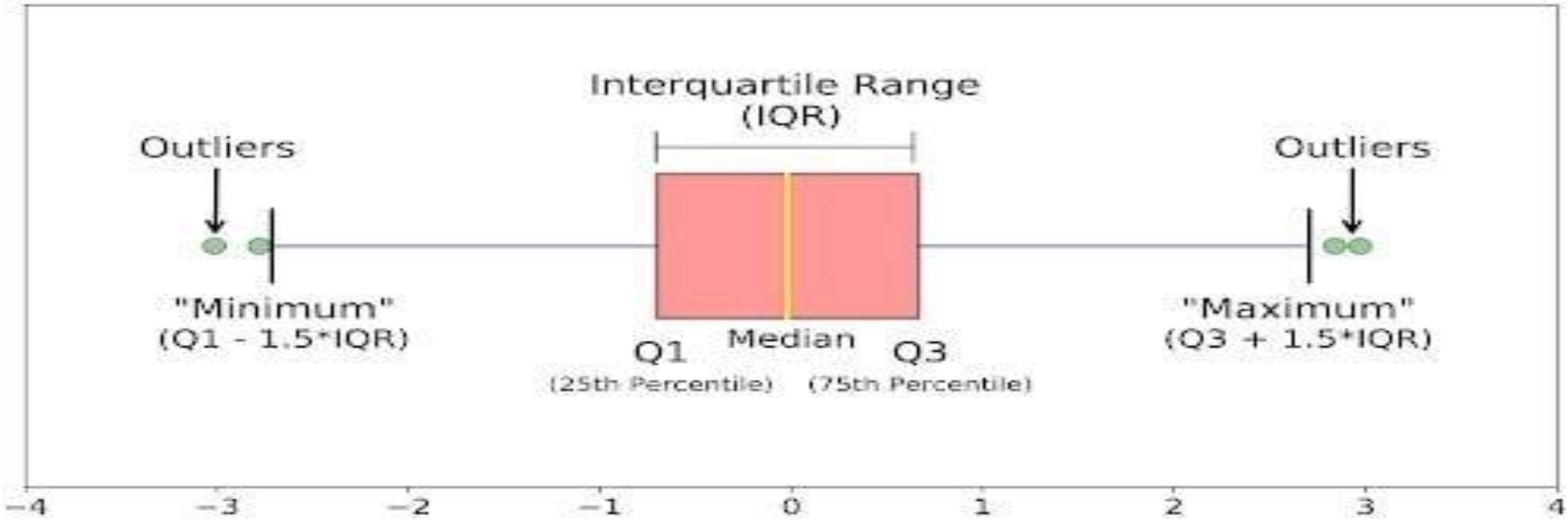
   - `plt.legend(title='Four Fruits:')`: Adds a legend to the plot with a title `'Four Fruits:'`. The legend explains the categories represented by each slice in the pie chart based on the labels provided in `mylabels`.

5. **Displaying the Plot:**

- `plt.show()`: Displays the pie chart on the screen. This function is necessary to render any Matplotlib plot interactively.

boxplot

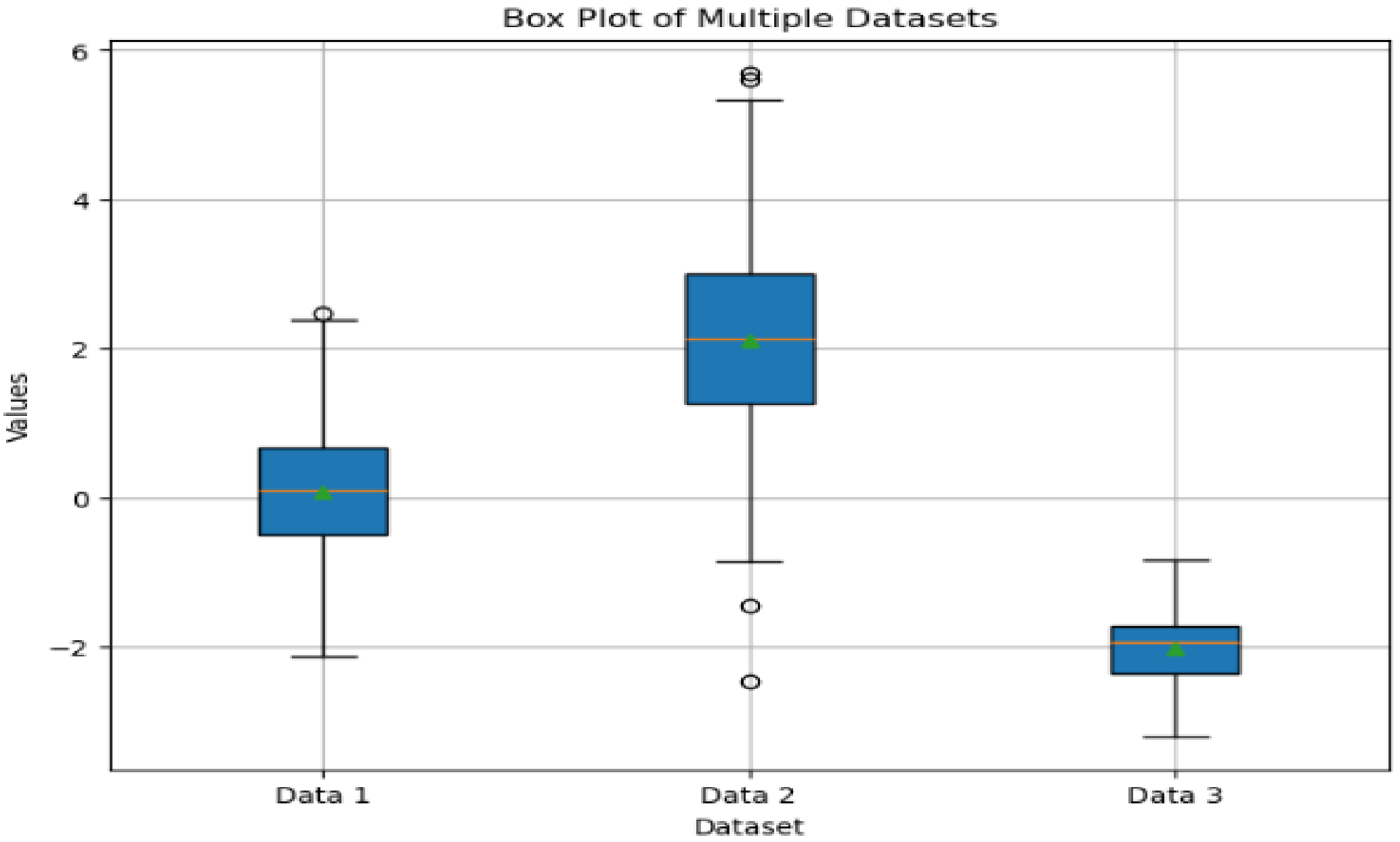
A boxplot, also known as a box-and-whisker plot, is a standardized way of displaying the distribution of numerical data based on a five-number summary: minimum, first quartile (Q1), median (Q2), third quartile (Q3), and maximum.
It is particularly useful for comparing distributions between different groups or



Characteristics of a Box Plot:

- **Box**: The box represents the interquartile range (IQR) which spans from the first quartile (Q1) to the third quartile (Q3). The median (Q2) is marked by a line inside the box.
- **Whiskers**: The whiskers extend from the edges of the box to show the range of the data, typically up to 1.5 times the IQR. Data points beyond this range are considered outliers and plotted individually.
- **Outliers**: Individual data points that lie beyond the whiskers are shown as dots or asterisks.

[1]:
```python
import matplotlib.pyplot as plt
import numpy as np

# Sample data
np.random.seed(10)
data1 = np.random.normal(loc=0, scale=1, size=100)
data2 = np.random.normal(loc=2, scale=1.5, size=100)
data3 = np.random.normal(loc=-2, scale=0.5, size=100)

# Create a box plot
plt.figure(figsize=(8, 6))
plt.boxplot([data1, data2, data3], labels=['Data 1', 'Data 2', 'Data 3'], patch_artist=True, showmeans=True)
plt.title('Box Plot of Multiple Datasets')
plt.xlabel('Dataset')
plt.ylabel('Values')
plt.grid(True)
plt.show()
```



1. Imports:

- `import matplotlib.pyplot as plt`: Imports the `matplotlib.pyplot` module under the alias `plt`, which provides a MATLAB-like plotting interface.
- `import numpy as np`: Imports the NumPy library under the alias `np`, which is used for numerical computations.

2. **Sample Data Generation:**

- `` `np.random.seed(10)` ``: Sets the random seed to 10 for reproducibility. This ensures that every time you run the code, the same random numbers are generated.
- `` `data1 = np.random.normal(loc=0, scale=1, size=100)` ``: Generates 100 random numbers from a normal distribution with mean (`` `loc` ``) 0 and standard deviation (`` `scale` ``) 1.
- `` `data2 = np.random.normal(loc=2, scale=1.5, size=100)` ``: Generates 100 random numbers from a normal distribution with mean 2 and standard deviation 1.5.
- `` `data3 = np.random.normal(loc=-2, scale=0.5, size=100)` ``: Generates 100 random numbers from a normal distribution with mean -2 and standard deviation 0.5.

3. Creating the Box Plot:

- `plt.figure(figsize=(8, 6))`: Creates a new figure with a specified figure size of width 8 inches and height 6 inches.
- `plt.boxplot([data1, data2, data3], labels=['Data 1', 'Data 2', 'Data 3'], patch_artist=True, showmeans=True)`:
  - `data1`, `data2`, and `data3` are the datasets passed as a list to create the box plot.
  - `labels=['Data 1', 'Data 2', 'Data 3']`: Labels for each dataset shown on the x-axis.
  - `patch_artist=True`: Fills the boxes of the box plot with colors.
  - `showmeans=True`: Shows the mean of each dataset as a green triangle inside the box.

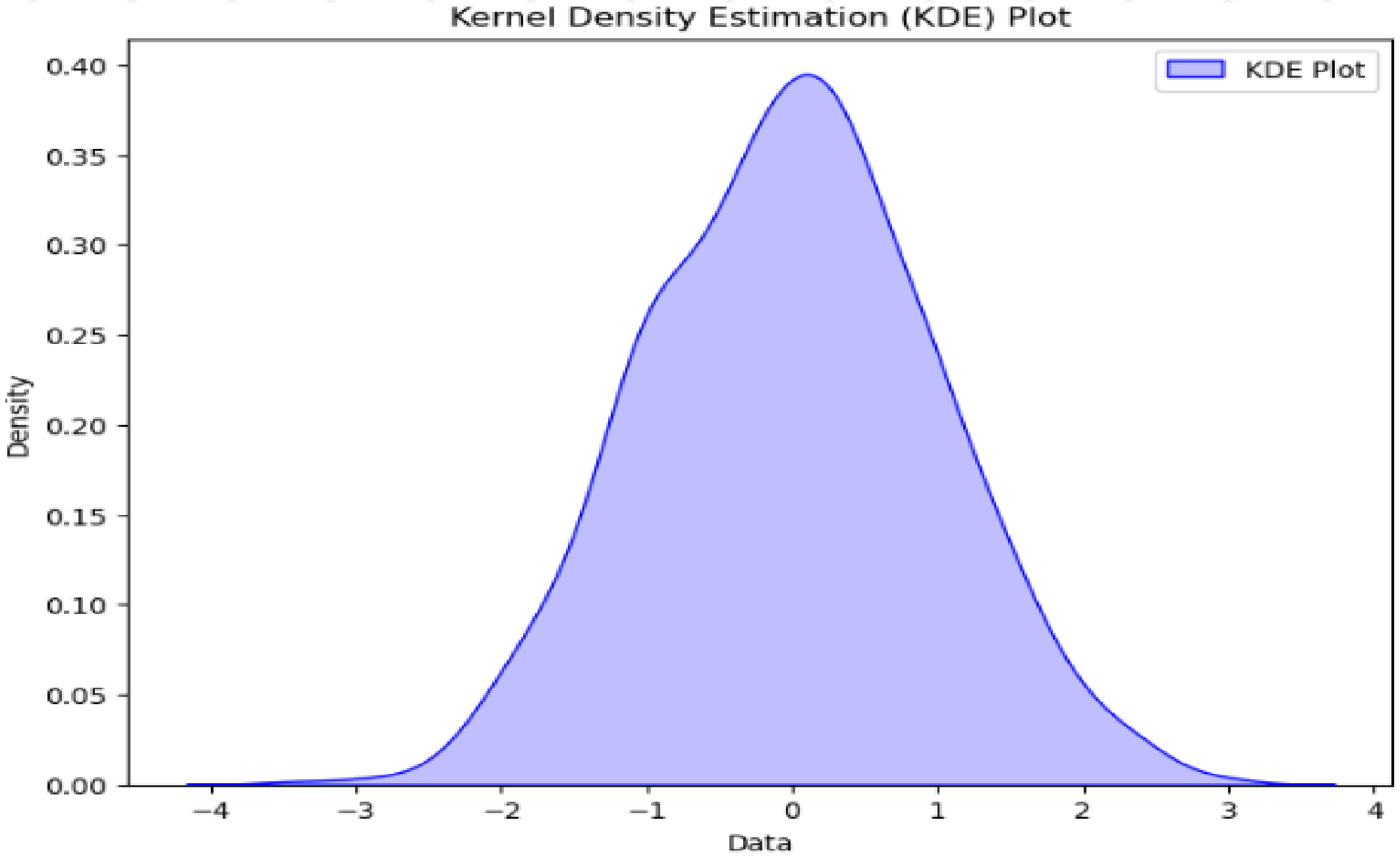
- `plt.title('Box Plot of Multiple Datasets')`: Sets the title of the plot.
- `plt.xlabel('Dataset')`: Sets the label for the x-axis.
- `plt.ylabel('Values')`: Sets the label for the y-axis.
- `plt.grid(True)`: Shows grid lines on the plot.
- `plt.show()`: Displays the plot.

Kernel Density Estimation

- The term "KDE" stands for Kernel Density Estimation.
- It is a technique used to estimate the probability density function (PDF) of a continuous random variable based on a finite data sample.
- KDE is particularly useful when you want to visualize the distribution of data points as a smooth curve rather than using discrete bins like in histograms.

[4]:

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

# Generate random data (normal distribution)
data = np.random.normal(loc=0, scale=1, size=1000)

# Create KDE plot using Seaborn
plt.figure(figsize=(8, 6))
sns.kdeplot(data, shade=True, color='b', label='KDE Plot')

# Add labels and title
plt.xlabel('Data')
plt.ylabel('Density')
plt.title('Kernel Density Estimation (KDE) Plot')

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



Code Summary:

1. Imports:

   - `import matplotlib.pyplot as plt`: Imports Matplotlib's pyplot module for creating plots.
   - `import seaborn as sns`: Imports Seaborn, a statistical data visualization library.
   - `import numpy as np`: Imports NumPy for generating random data.

2. Generate Random Data:

   - `data = np.random.normal(loc=0, scale=1, size=1000)`: Generates 1000 random numbers from a normal distribution with mean (`loc`) 0 and standard deviation (`scale`) 1.

3. Create KDE Plot using Seaborn:

- `plt.figure(figsize=(8, 6))`: Creates a new figure with a size of 8 inches (width) by 6 inches (height).
- `sns.kdeplot(data, shade=True, color='b', label='KDE Plot')`: Creates a KDE plot using Seaborn (`sns.kdeplot`).
  - `data`: Specifies the data array to be plotted.
  - `shade=True`: Fills the area under the KDE curve.
  - `color='b'`: Sets the color of the KDE curve to blue ('b').
  - `label='KDE Plot'`: Specifies the label for the plot legend.

4. **Customize Labels and Title:**

   - `plt.xlabel('Data')`: Adds a label to the x-axis, describing the data values.
   - `plt.ylabel('Density')`: Adds a label to the y-axis, indicating the density of the data distribution.
   - `plt.title('Kernel Density Estimation (KDE) Plot')`: Adds a title to the plot, providing context for the visualization.

5. **Display the Plot:**

   - `plt.legend()`: Displays the legend for the plot (based on the `label` parameter in `sns.kdeplot`).
   - `plt.show()`: Displays the KDE plot in the Python environment.

Violin Plot

- A violin plot is a type of data visualization that combines elements of a box plot and a kernel density plot.
- It is used to visualize the distribution and density of data across different levels or categories.

Characteristics of a Violin Plot:

1. **Shape**: The violin plot resembles a violin or a mirrored density plot with a box plot inside. It shows the distribution of data along a continuous variable or grouped by a categorical variable.

2. **Components**:

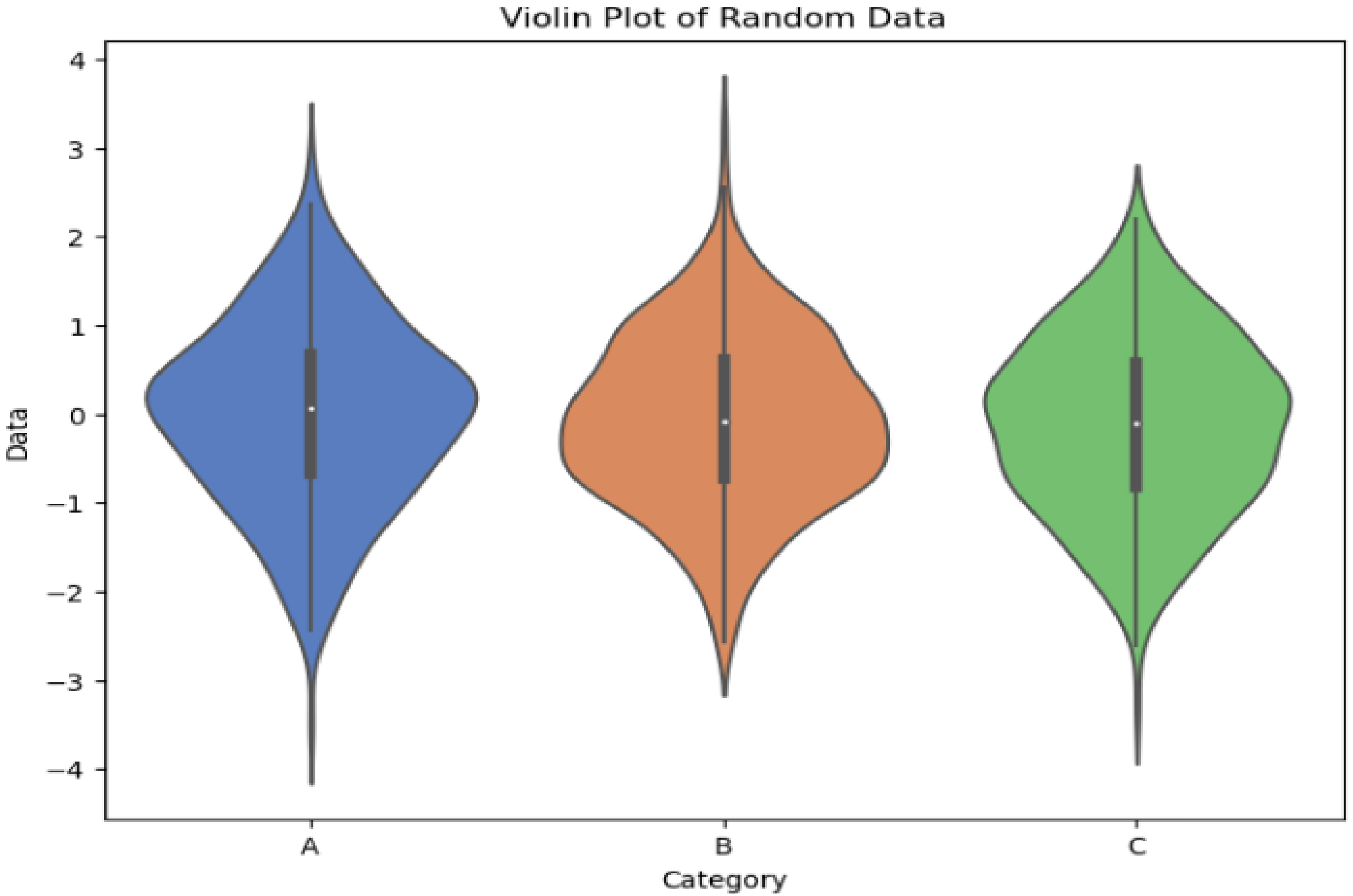
   - **Box Plot**: The central box represents the interquartile range (IQR) and median of the data.
   - **Density Plot**: The density of the data at different values is represented by the width of the "violin" shape. Wider areas indicate higher data density.

3. **Interpretation**:

   - **Width**: The width at any given point along the violin indicates the probability density of the data points.
   - **Symmetry**: Violin plots can show asymmetry or skewness in the data distribution.
   - **Outliers**: Outliers are shown as points outside the "violin" shape.

[3]:

```python
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
import pandas as pd
# Generate random data (normal distribution)
data = np.random.normal(loc=0, scale=1, size=1000)
categories = np.random.choice(['A', 'B', 'C'], size=1000)

# Create a DataFrame for seaborn
df = pd.DataFrame({'Data': data, 'Category': categories})

# Create violin plot
plt.figure(figsize=(8, 6))
sns.violinplot(x='Category', y='Data', data=df, palette='muted')

# Add labels and title
plt.xlabel('Category')
plt.ylabel('Data')
plt.title('Violin Plot of Random Data')

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



Code Summary:

1. Imports:
    - `import matplotlib.pyplot as plt`: Imports Matplotlib's pyplot module for creating plots.
    - `import seaborn as sns`: Imports Seaborn, a statistical data visualization library.
    - `import numpy as np`: Imports NumPy for generating random data.
    - `import pandas as pd`: Imports Pandas for creating DataFrames.
2. Generate Random Data:
    - `data = np.random.normal(loc=0, scale=1, size=1000)`: Generates 1000 random numbers from a normal distribution with mean (`loc`) 0 and standard deviation (`scale`) 1.
    - `categories = np.random.choice(['A', 'B', 'C'], size=1000)`: Generates 1000 random categories ('A', 'B', 'C').

3. Create DataFrame for Seaborn:

- `df = pd.DataFrame({'Data': data, 'Category': categories})`: Creates a Pandas DataFrame from `data` and `categories`, where `'Data'` represents the numerical values and `'Category'` represents the categorical groups.

4. Create Violin Plot:

- `plt.figure(figsize=(8, 6))`: Creates a new figure with a size of 8 inches (width) by 6 inches (height).
- `sns.violinplot(x='Category', y='Data', data=df, palette='muted')`: Creates a violin plot using Seaborn (`sns.violinplot`).
  - `x='Category'`: Specifies the categorical variable for the x-axis (groups).
  - `y='Data'`: Specifies the numerical variable for the y-axis (values to be plotted).
  - `data=df`: Specifies the DataFrame containing the data (`df`).
  - `palette='muted'`: Specifies the color palette for the plot.

5. Customize Labels and Title:

- `plt.xlabel('Category')`: Adds a label to the x-axis, describing the categories.
- `plt.ylabel('Data')`: Adds a label to the y-axis, describing the data values.
- `plt.title('Violin Plot of Random Data')`: Adds a title to the plot, providing context for the visualization.

6. Display the Plot:

- `plt.show()`: Displays the violin plot in the Python environment.

Heatmap

Characteristics of Heatmaps:

1. **Color Representation**: Each value in the matrix is assigned a color, typically on a gradient scale, where colors represent different ranges of values. This allows for quick visual identification of high and low values.

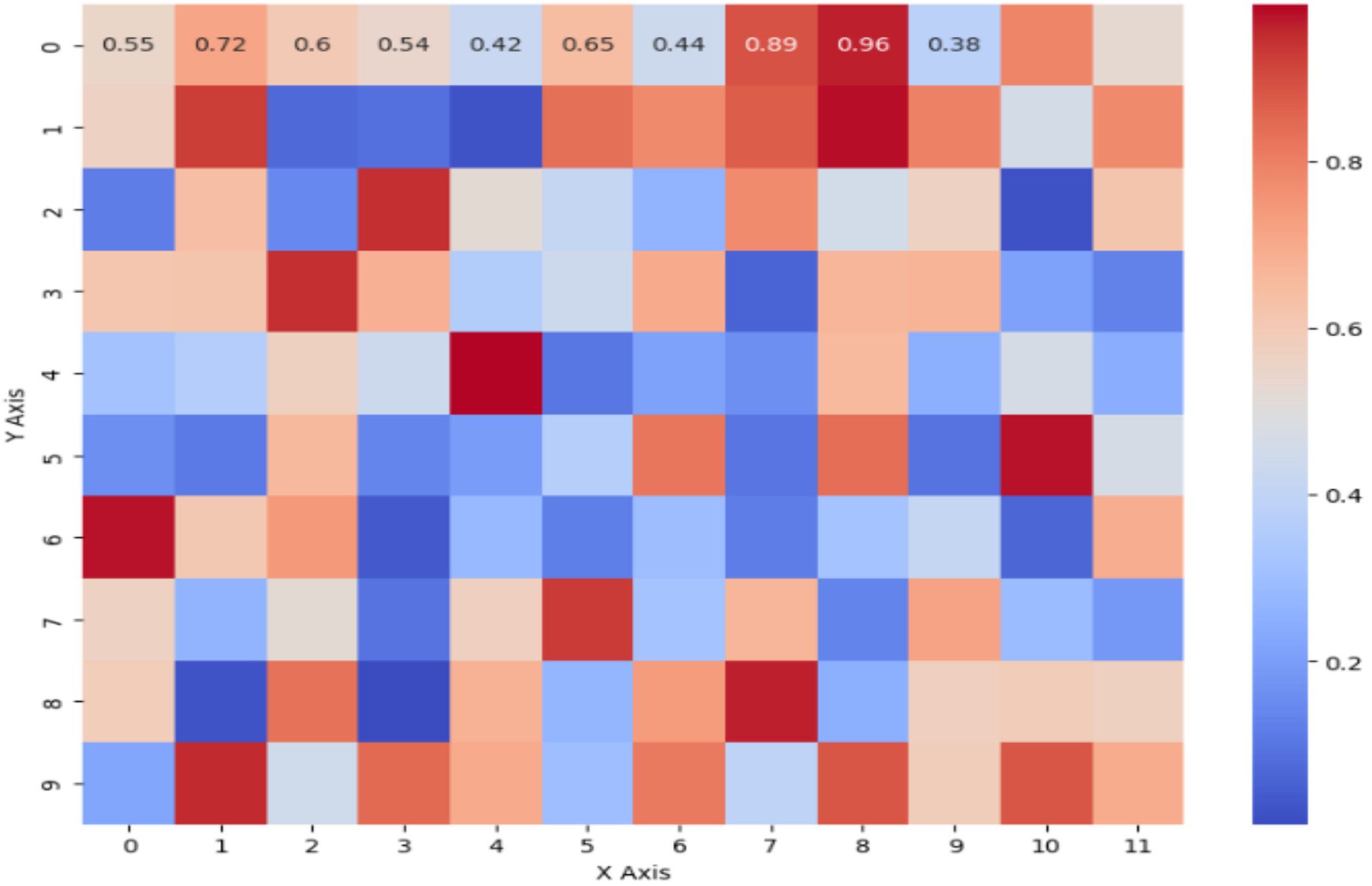
2. **Matrix Visualization**: Heatmaps are often used to display correlation matrices, where each cell represents the correlation coefficient between two variables.

3. **Applications**:

   - **Data Analysis**: Heatmaps help in identifying trends, clusters, and outliers within datasets.
   - **Heatmap Types**: There are different types of heatmaps, including hierarchical clustering heatmaps and correlation heatmaps.

[4]:

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

# Generate sample data (correlation matrix)
np.random.seed(0)
data = np.random.rand(10, 12)

# Create heatmap
plt.figure(figsize=(10, 8))
sns.heatmap(data, annot=True, cmap='coolwarm')
plt.title('Heatmap of Random Data')
plt.xlabel('X Axis')
plt.ylabel('Y Axis')
plt.show()
```


Heatmap of Random Data
0.55
0.72
0.6
0.54
0.42
0.65
0.44
0.89
0.96
0.38
Y Axis
X Axis
0
1
2
3
4
5
6
7
8
9
10
11
0.8
0.6
0.4
0.2

- Step 1: Import Libraries
  - `import matplotlib.pyplot as plt`: Imports Matplotlib for plotting.
  - `import seaborn as sns`: Imports Seaborn for statistical data visualization.
  - `import numpy as np`: Imports NumPy for numerical computations.
- Step 2: Generate Sample Data
  - `np.random.seed(0)`: Sets the random seed for reproducibility.
  - `data = np.random.rand(10, 12)`: Generates a 10x12 matrix of random numbers between 0 and 1. This could represent, for example, a correlation matrix or any other matrix of interest.

- Step 3: Create Heatmap
  - `plt.figure(figsize=(10, 8))`: Creates a new figure with a specified size (width=10 inches, height=8 inches).
  - `sns.heatmap(data, annot=True, cmap='coolwarm')`: Creates the heatmap using Seaborn's `heatmap` function. Parameters include:
    - `data`: The input data (the matrix to be visualized).
    - `annot=True`: Displays the numerical values in each cell.
    - `cmap='coolwarm'`: Sets the color map for the heatmap.
  - `plt.title`, `plt.xlabel`, `plt.ylabel`: Adds title and labels to the plot.
  - `plt.show()`: Displays the heatmap.

Creating a word cloud is a popular way to visualize the frequency of words in a text dataset. `matplotlib` can be used in conjunction with the `wordcloud` library to generate visually appealing word clouds. Here's a step-by-step guide on how to create a word cloud using these libraries:

Installation:

First, you need to install the necessary libraries if you haven't already:

bash Copy code

```
pip install matplotlib wordcloud
```

1. **Import Libraries:**

```python
import matplotlib.pyplot as plt
from wordcloud import WordCloud
```

2. **Prepare Text Data:**

   Suppose you have a string containing text data:

```python
text = "Python is a programming language. It is widely used for data analysis and mach
```

3. **Generate Word Cloud:**

Use the `WordCloud` class to generate a word cloud from the text data:

```python
# Generate word cloud
wordcloud = WordCloud(width=800, height=400, background_color='white').generate(text)
```

Here:

- `width` and `height` specify the dimensions of the word cloud image.
- `background_color` sets the background color of the word cloud image (default is `'black'`).

4. **Display the Word Cloud**:

Use `matplotlib` to display the generated word cloud:

python Copy code

```
# Display the word cloud using matplotlib
plt.figure(figsize=(10, 5))
plt.imshow(wordcloud, interpolation='bilinear')
plt.axis('off')  # Turn off axis numbers and ticks
plt.show()
```

- `plt.figure(figsize=(10, 5))` sets the size of the figure.
- `plt.imshow(wordcloud, interpolation='bilinear')` displays the word cloud with bilinear interpolation for smoother edges.
- `plt.axis('off')` removes the axis numbers and ticks for a cleaner appearance.
- `plt.show()` displays the plot.

[103]:
```
pip install WordCloud
```

```
Requirement already satisfied: WordCloud in c:\users\gunti\anaconda3\lib\site-packages (1.9.3)
Requirement already satisfied: numpy>=1.6.1 in c:\users\gunti\anaconda3\lib\site-packages (from WordCloud) (1.26.4)
Requirement already satisfied: pillow in c:\users\gunti\anaconda3\lib\site-packages (from WordCloud) (10.2.0)
Requirement already satisfied: matplotlib in c:\users\gunti\anaconda3\lib\site-packages (from WordCloud) (3.8.0)
Requirement already satisfied: contourpy>=1.0.1 in c:\users\gunti\anaconda3\lib\site-packages (from matplotlib->WordCloud) (1.2.0)
Requirement already satisfied: cycler>=0.10 in c:\users\gunti\anaconda3\lib\site-packages (from matplotlib->WordCloud) (0.11.0)
Requirement already satisfied: fonttools>=4.22.0 in c:\users\gunti\anaconda3\lib\site-packages (from matplotlib->WordCloud) (4.25.0)
Requirement already satisfied: kiwisolver>=1.0.1 in c:\users\gunti\anaconda3\lib\site-packages (from matplotlib->WordCloud) (1.4.4)
Requirement already satisfied: packaging>=20.0 in c:\users\gunti\anaconda3\lib\site-packages (from matplotlib->WordCloud) (23.1)
Requirement already satisfied: pyparsing>=2.3.1 in c:\users\gunti\anaconda3\lib\site-packages (from matplotlib->WordCloud) (3.0.9)
Requirement already satisfied: python-dateutil>=2.7 in c:\users\gunti\anaconda3\lib\site-packages (from matplotlib->WordCloud) (2.8.2)
Requirement already satisfied: six>=1.5 in c:\users\gunti\anaconda3\lib\site-packages (from python-dateutil>=2.7->matplotlib->WordCloud) (1.16.0)
Note: you may need to restart the kernel to use updated packages.
```

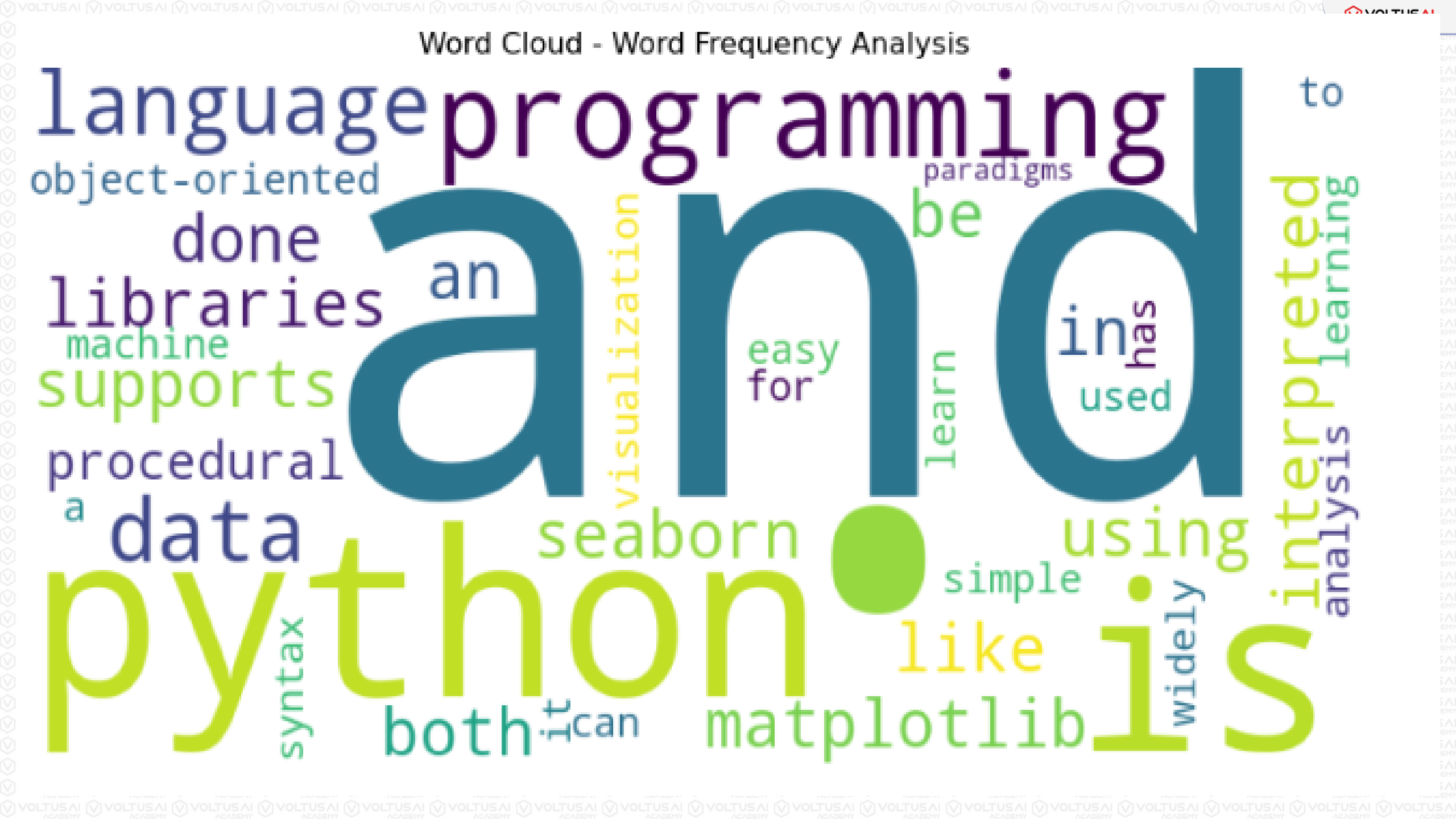
[105]:
```
import nltk
from nltk.tokenize import word_tokenize
from nltk.probability import FreqDist
import pandas as pd
import matplotlib.pyplot as plt
from wordcloud import WordCloud

# Example text data
text = """
Python is a programming language. It is widely used for data analysis and machine learning.
Python has simple syntax and is easy to learn. Data visualization in Python can be done using
libraries like Matplotlib and Seaborn. Python is an interpreted language and supports both
procedural and object-oriented programming paradigms.
"""

# Tokenize the text
words = word_tokenize(text.lower())


print(words)
```

```
['python', 'is', 'a', 'programming', 'language', '.', 'it', 'is', 'widely', 'used', 'for', 'data', 'analysis', 'and', 'machine', 'learning', '.', 'pytho
n', 'has', 'simple', 'syntax', 'and', 'is', 'easy', 'to', 'learn', '.', 'data', 'visualization', 'in', 'python', 'can', 'be', 'done', 'using', 'librarie
s', 'like', 'matplotlib', 'and', 'seaborn', '.', 'python', 'is', 'an', 'interpreted', 'language', 'and', 'supports', 'both', 'procedural', 'and', 'object
-oriented', 'programming', 'paradigms', '.']
```

[106]:
```
# Create a frequency distribution of words
freq_dist = FreqDist(words)

# Convert to pandas DataFrame
freq_df = pd.DataFrame(list(freq_dist.items()), columns=['Word', 'Frequency'])

# Sort DataFrame by 'Frequency' column in descending order
freq_df_sorted = freq_df.sort_values(by='Frequency', ascending=False)

# Display the sorted DataFrame
print("Word Frequency Analysis (Sorted in Descending Order):")
print(freq_df_sorted)
print()

# Generate word cloud from sorted frequencies
wordcloud = WordCloud(width=800, height=400, background_color='white').generate_from_frequencies(dict(freq_df_sorted.values))

# Display the word cloud using matplotlib
plt.figure(figsize=(10, 6))
plt.imshow(wordcloud, interpolation='bilinear')
plt.axis('off')
plt.title('Word Cloud - Word Frequency Analysis')
plt.show()
```

Word Frequency Analysis (Sorted in Descending Order):

	Word	Frequency
5	.	5
12	and	5
0	python	4
1	is	4
3	programming	2
4	language	2
10	data	2
29	matplotlib	1
24	be	1
25	done	1
26	using	1
27	libraries	1
28	like	1
31	an	1
30	seaborn	1
22	in	1
32	interpreted	1
33	supports	1
34	both	1
35	procedural	1
36	object-oriented	1
23	can	1
19	to	1
21	visualization	1
20	learn	1
18	easy	1
17	syntax	1
16	simple	1
15	has	1
14	learning	1
13	machine	1
11	analysis	1
9	for	1
8	used	1
7	widely	1
6	it	1
2	a	1


Word Cloud - Word Frequency Analysis
language
programming
to
object-oriented
paradigms
done
be
an
visualization
and
libraries
machine
supports
easy
for
learn
in
has
used
interpreted
learning
analysis
procedural
a
data
seaborn
using
python
simple
syntax
like
widely
is
both
it
can
matplotlib

Default datasets available in popular libraries such as scikit-learn, seaborn, and pandas.

1. Scikit-learn Datasets

Scikit-learn provides several built-in datasets that are useful for practicing machine learning algorithms. You can load these datasets using `sklearn.datasets`. Here are some of the key datasets:

- **Iris Dataset:** Used for classification tasks. Contains measurements of iris flowers from three different species.
- **Digits Dataset:** Contains images of handwritten digits, used for classification.
- **Wine Dataset:** Used for classification, contains chemical analysis results of wines from three different cultivars.
- **Breast Cancer Dataset:** Used for binary classification, contains features for predicting breast cancer diagnoses.
- **Diabetes Dataset:** Used for regression tasks, contains medical measurements and a target variable for diabetes progression.

- **Boston Housing Dataset**: Contains information on housing prices in Boston (deprecated in newer versions; use `fetch_california_housing` instead).

2. Seaborn Datasets

Seaborn, a statistical data visualization library, also includes several datasets that are useful for demonstration and practice. You can load these datasets using `seaborn.load_dataset()`. Some notable datasets are:

- **Titanic**: Information about passengers on the Titanic, used for classification and survival analysis.
- **Penguins**: Contains measurements of penguins, used for classification and exploratory data analysis.
- **Tips**: Contains data on tips given in a restaurant, useful for regression and categorical analysis.
- **Flights**: Contains flight data, useful for time-series analysis and visualization.

- The **diabetes dataset** is commonly used in machine learning and data analysis to predict diabetes progression based on various features.
- It's often used with libraries like Scikit-Learn for modeling, but it can also be visualized using Matplotlib.

Features:

The dataset consists of 10 features that are related to diabetes, including:

1. **Age**: Age of the patient.
2. **Sex**: Gender of the patient.
3. **BMI**: Body mass index.
4. **Blood Pressure**: Average blood pressure.
5. **S1**: Total cholesterol.
6. **S2**: Low-density lipoprotein cholesterol.
7. **S3**: High-density lipoprotein cholesterol.
8. **S4**: Total cholesterol / high-density lipoprotein cholesterol.
9. **S5**: Low-density lipoprotein cholesterol / total cholesterol.
10. **S6**: High-density lipoprotein cholesterol / low-density lipoprotein cholesterol.

Target Variable:

- **Target:** A quantitative measure of disease progression one year after baseline, represented as a continuous value.

Dataset Characteristics:

- Number of Instances: 442
- Number of Features: 10
- Type of Problem: Regression

Preprocessing of the Diabetes Dataset

1. **Normalization**:
    - All features (attributes) in the diabetes dataset are preprocessed through a process known as standardization or normalization.
    - This process scales each feature to have a mean of 0 and a standard deviation of 1 across all samples.
2. **Standardization Process**:
    - Each attribute X in the dataset is transformed using the formula:

$$X_{\text{std}} = \frac{X - \bar{X}}{\sigma_X}$$

where $\bar{X}$ is the mean of X across all samples, and σ_X is the standard deviation of X.

3. **Purpose of Standardization**:

   - **Equal Contribution**: Ensures that all features contribute equally to the analysis and model fitting process.
   - **Algorithm Performance**: Helps machine learning algorithms converge faster and prevents features with larger numerical ranges from dominating those with smaller ranges.
   - **Model Interpretation**: Simplifies the interpretation of model coefficients or feature importance.

4. **Attributes in the Diabetes Dataset**:

   - Each attribute in the dataset represents a quantitative measure related to diabetes health, such as age, body mass index (BMI), blood pressure (BP), and others.

[1]:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.datasets import load_diabetes

# Load the diabetes dataset
diabetes = load_diabetes()

# Convert to DataFrame
df = pd.DataFrame(data=diabetes.data, columns=diabetes.feature_names)
df['target'] = diabetes.target

# Display the first few rows
print(df.head())
```

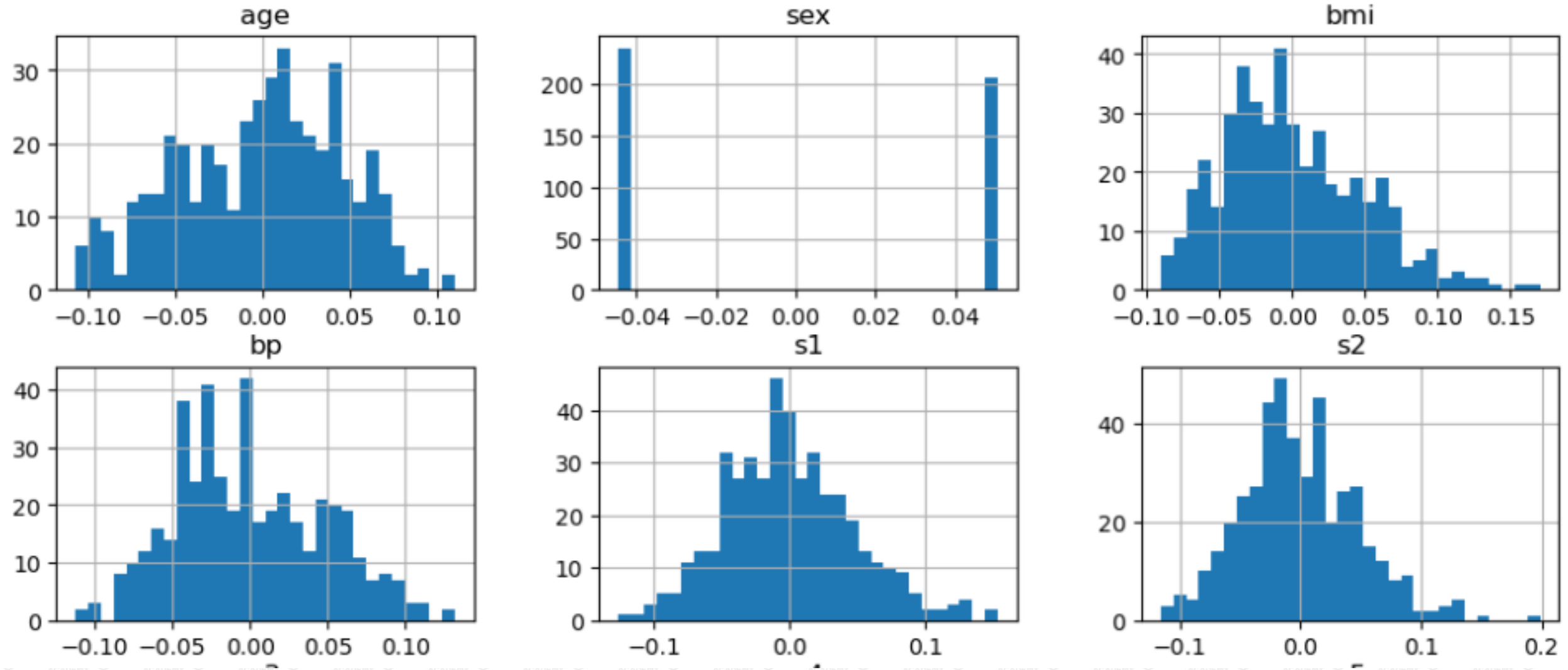
	age	sex	bmi	bp	s1	s2	s3	\
0	0.038076	0.050680	0.061696	0.021872	-0.044223	-0.034821	-0.043401	
1	-0.001882	-0.044642	-0.051474	-0.026328	-0.008449	-0.019163	0.074412	
2	0.085299	0.050680	0.044451	-0.005670	-0.045599	-0.034194	-0.032356	
3	-0.089063	-0.044642	-0.011595	-0.036656	0.012191	0.024991	-0.036038	
4	0.005383	-0.044642	-0.036385	0.021872	0.003935	0.015596	0.008142	

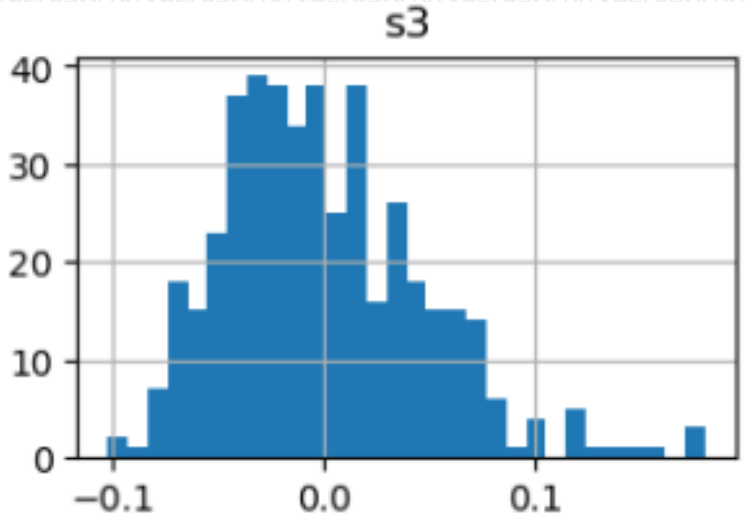
	s4	s5	s6	target
0	-0.002592	0.019907	-0.017646	151.0
1	-0.039493	-0.068332	-0.092204	75.0
2	-0.002592	0.002861	-0.025930	141.0
3	0.034309	0.022688	-0.009362	206.0
4	-0.002592	-0.031988	-0.046641	135.0

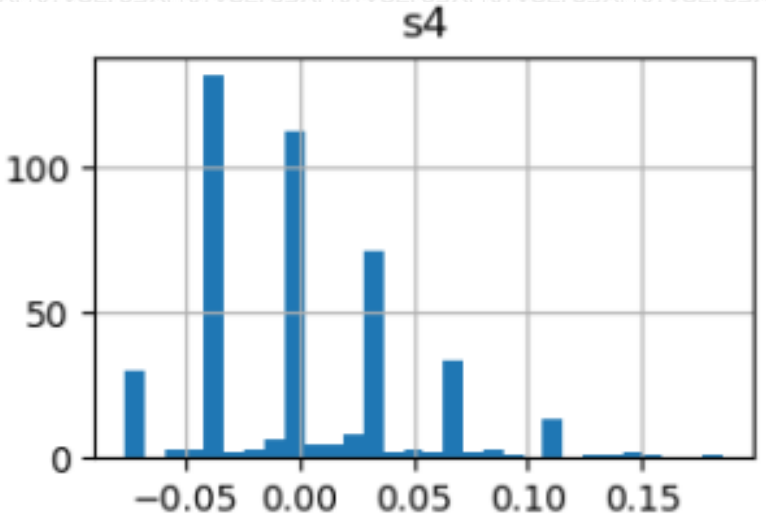
2. Basic Exploration and Visualization

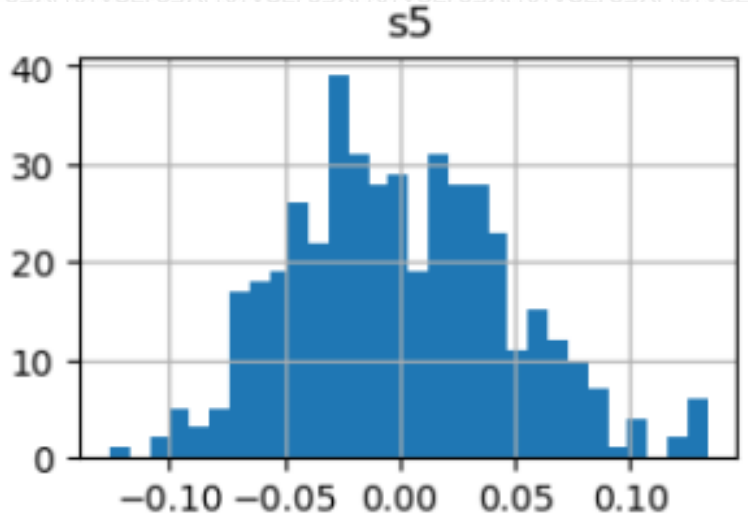
[2]:

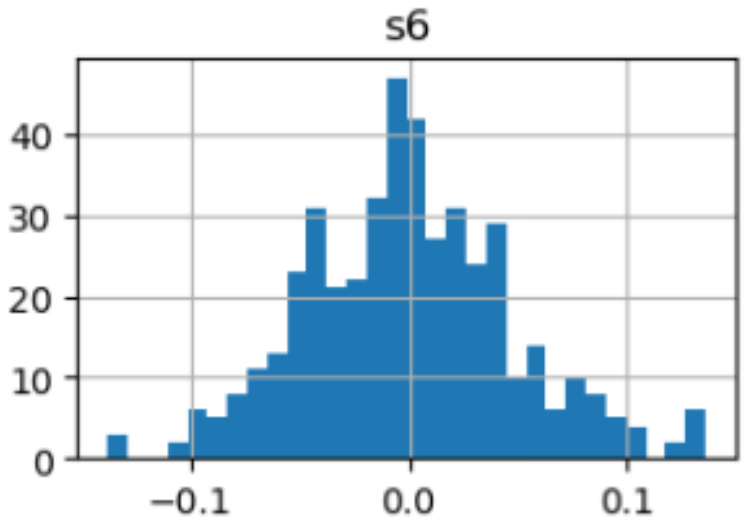
```
# Plot histograms for each feature
df.drop('target', axis=1).hist(bins=30, figsize=(12, 10))
plt.suptitle('Histograms of Diabetes Dataset Features')
plt.show()

# Plot histogram for target variable
plt.figure(figsize=(8, 6))
plt.hist(df['target'], bins=30, color='skyblue', edgecolor='black')
plt.title('Histogram of Diabetes Progression')
plt.xlabel('Disease Progression')
plt.ylabel('Frequency')
plt.show()
```

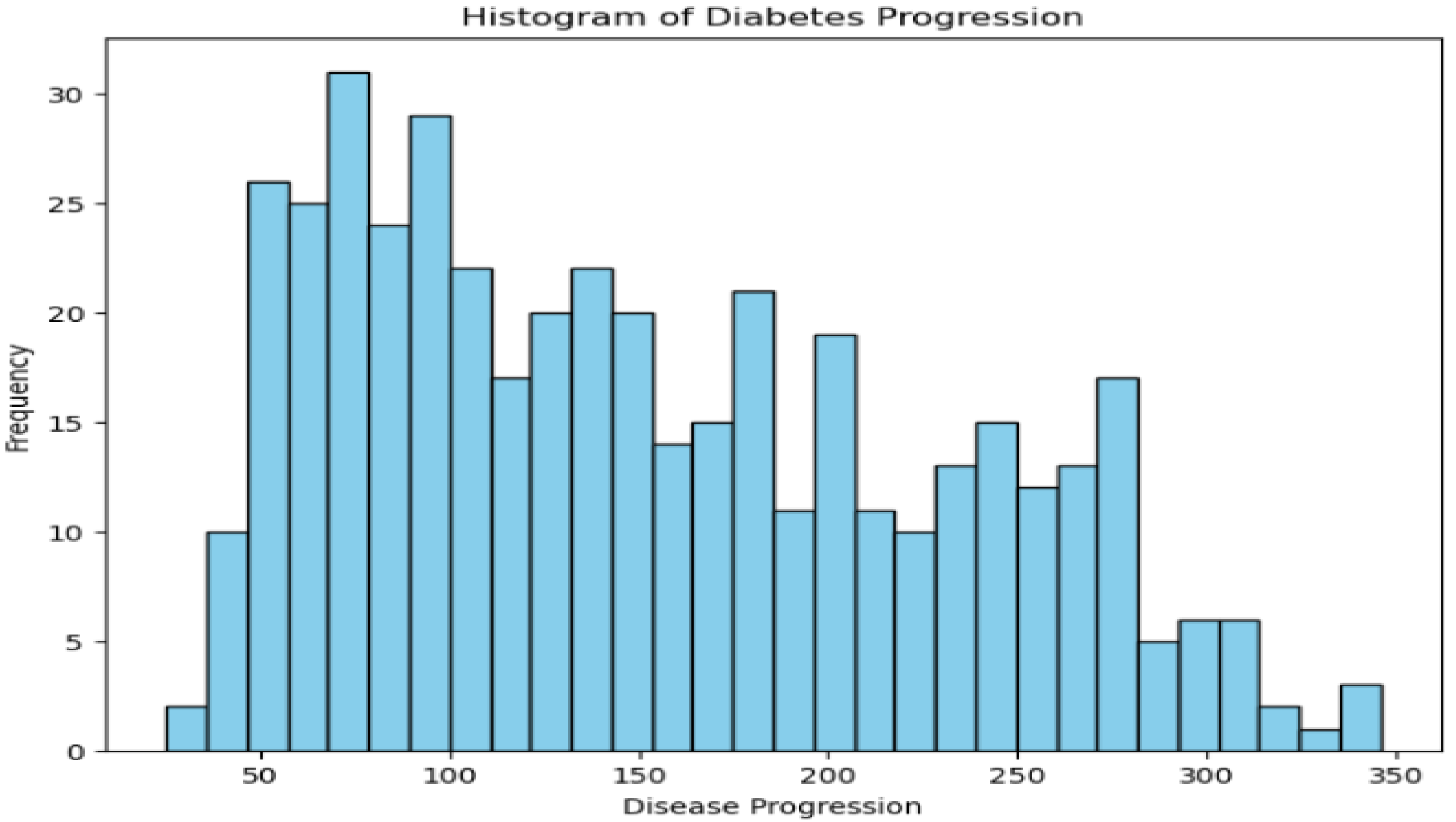



s3
40
30
20
10
0
−0.1
0.0
0.1


s4
100
50
0
−0.05
0.00
0.05
0.10
0.15

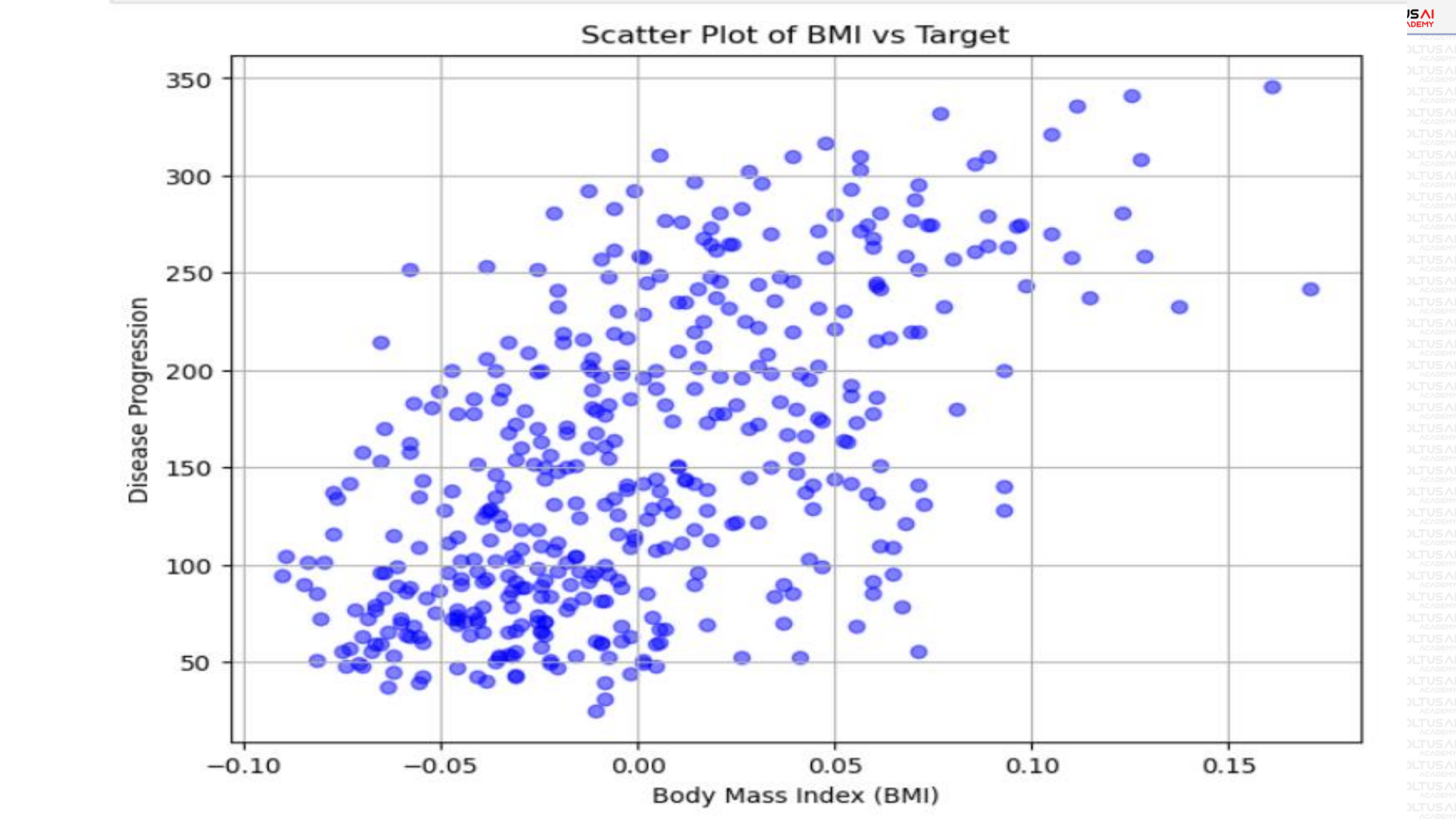

s5
40
30
20
10
0
−0.10
−0.05
0.00
0.05
0.10


s6
40
30
20
10
0
−0.1
0.0
0.1


Histogram of Diabetes Progression
Frequency
Disease Progression
0
5
10
15
20
25
30
50
100
150
200
250
300
350

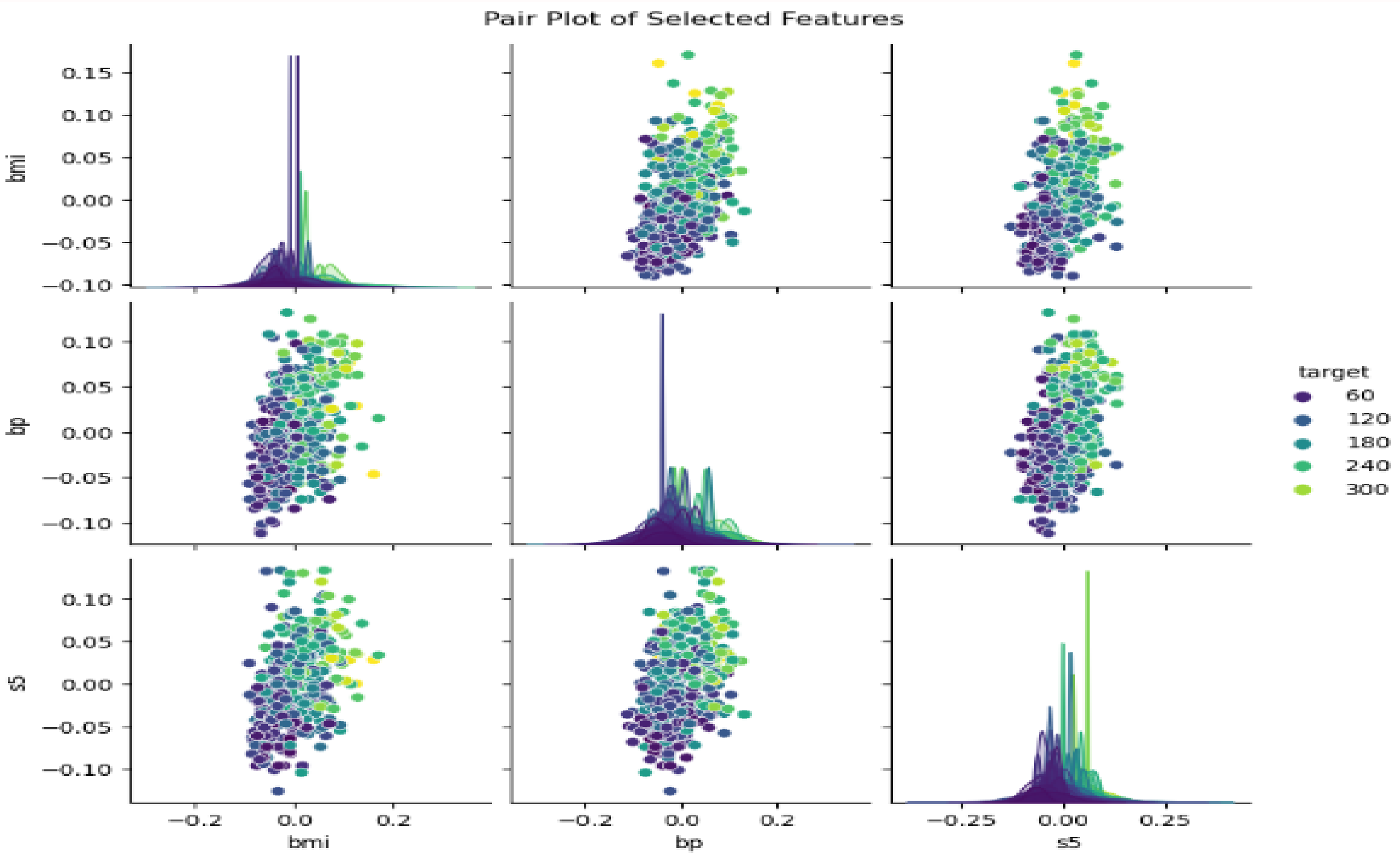
[3]:

```
# Scatter plot for 'bmi' feature vs target
plt.figure(figsize=(8, 6))
plt.scatter(df['bmi'], df['target'], alpha=0.5, color='blue')
plt.title('Scatter Plot of BMI vs Target')
plt.xlabel('Body Mass Index (BMI)')
plt.ylabel('Disease Progression')
plt.grid(True)
plt.show()
```


Scatter Plot of BMI vs Target
Disease Progression
350
300
250
200
150
100
50
−0.10
−0.05
0.00
0.05
0.10
0.15
Body Mass Index (BMI)

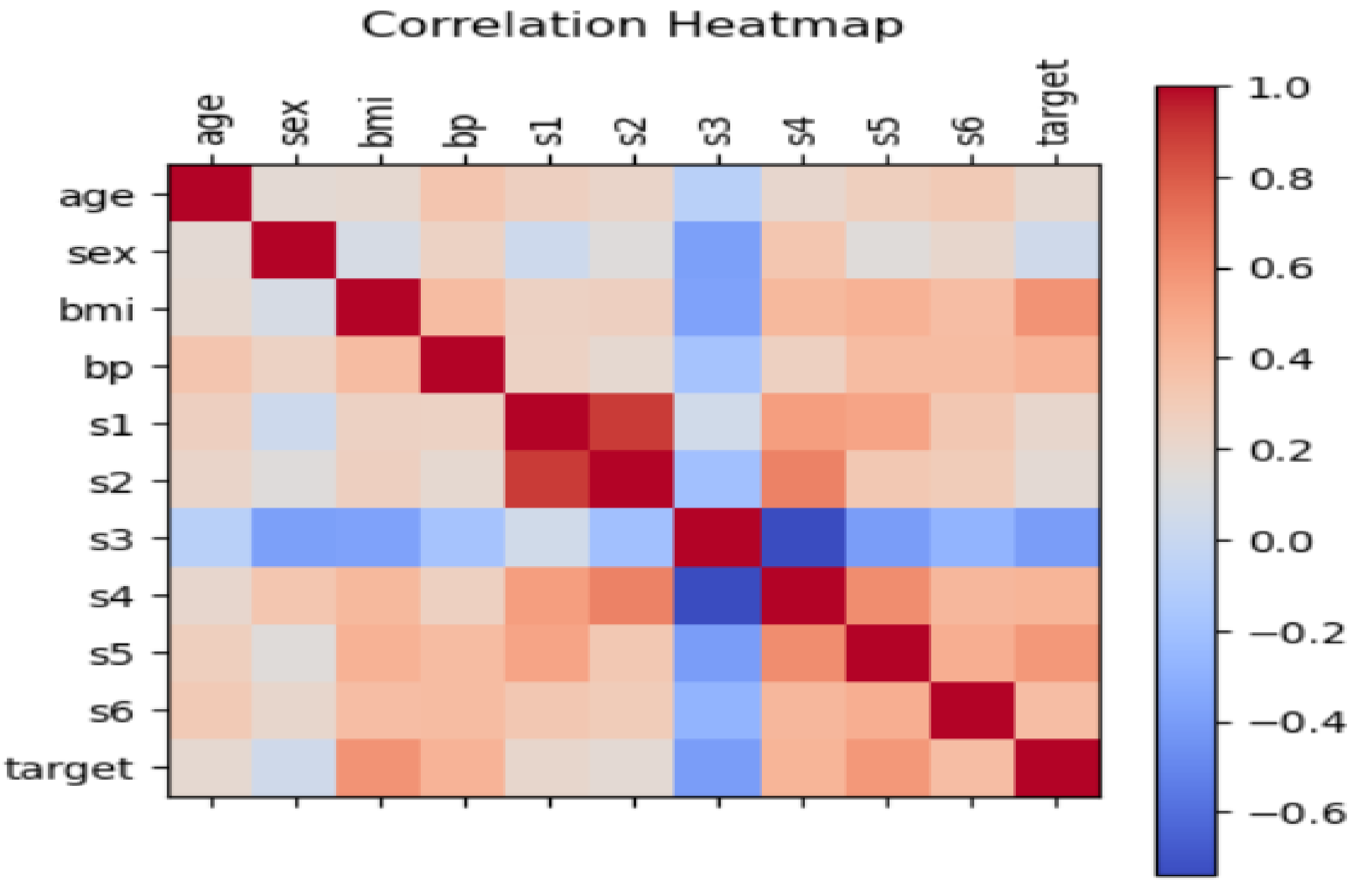
[4]:

```
import seaborn as sns


# Create pair plots for a subset of features
subset = df[['bmi', 'bp', 's5', 'target']]
sns.pairplot(subset, hue='target', palette='viridis')
plt.suptitle('Pair Plot of Selected Features', y=1.02)
plt.show()
```

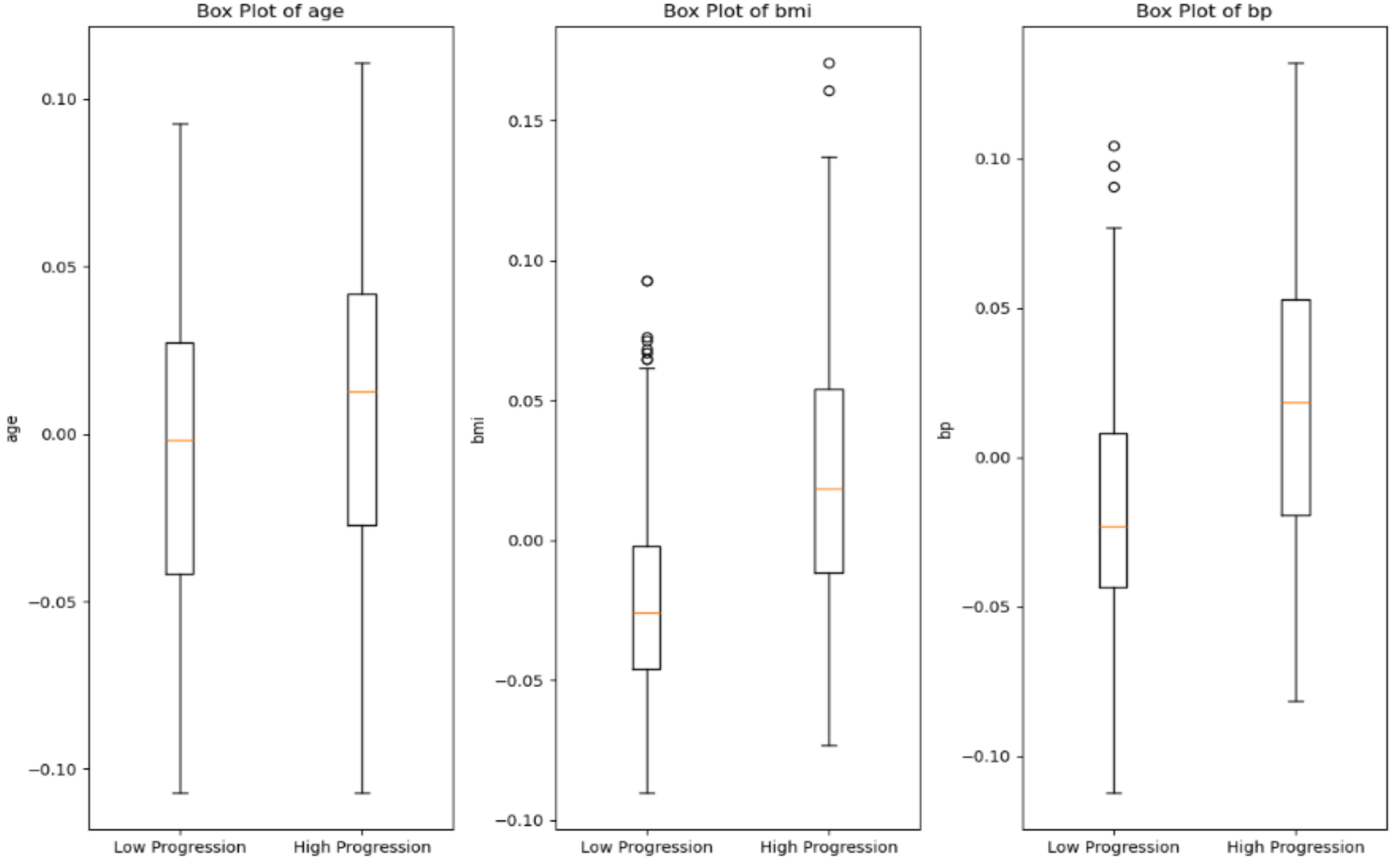

Pair Plot of Selected Features
bmi
bp
s5
0.15
0.10
0.05
0.00
−0.05
−0.10
−0.2
0.0
0.2
−0.25
0.00
0.25
target
60
120
180
240
300

[5]:

```
# Compute the correlation matrix
corr = df.corr()

# Create a heatmap of the correlation matrix
plt.figure(figsize=(12, 10))
cax = plt.matshow(corr, cmap='coolwarm')
plt.colorbar(cax)
plt.title('Correlation Heatmap')
plt.xticks(ticks=np.arange(len(corr.columns)), labels=corr.columns, rotation=90)
plt.yticks(ticks=np.arange(len(corr.columns)), labels=corr.columns)
plt.show()
```


Correlation Heatmap
age
sex
bmi
bp
s1
s2
s3
s4
s5
s6
target
1.0
0.8
0.6
0.4
0.2
0.0
−0.2
−0.4
−0.6

[6]:
```
# Create box plots for a few features
features = ['age', 'bmi', 'bp']

plt.figure(figsize=(12, 8))
for i, feature in enumerate(features):
    plt.subplot(1, len(features), i + 1)
    plt.boxplot([df[df['target'] < df['target'].median()][feature],
                 df[df['target'] >= df['target'].median()][feature]],
                labels=['Low Progression', 'High Progression'])
    plt.title(f'Box Plot of {feature}')
    plt.ylabel(feature)
plt.tight_layout()
plt.show()
```


Box Plot of age
Box Plot of bmi
Box Plot of bp
age
bmi
bp
0.10
0.05
0.00
−0.05
−0.10
0.15
Low Progression
High Progression

Summary

- **Histograms** show the distribution of individual features and the target variable.
- **Scatter plots** reveal relationships between pairs of features or between a feature and the target.
- **Pair plots** are useful for exploring pairwise relationships between features.
- **Correlation heatmaps** display the strength and direction of relationships between features and the target.
- **Box plots** provide insight into the distribution of features with respect to different levels of the target variable.

- The Titanic dataset is a classic dataset in the field of data science and machine learning.
- It contains information about passengers aboard the Titanic, including whether they survived or not, their age, sex, ticket class, fare paid, cabin, and more.
- This dataset is often used for teaching and practicing data analysis and prediction techniques, particularly for predicting survival based on other factors.

1. **PassengerId**: Unique identifier for each passenger.
2. **Survived**: Whether the passenger survived (0 = No, 1 = Yes).
3. **Pclass**: Ticket class (1 = 1st, 2 = 2nd, 3 = 3rd).
4. **Name**: Passenger's name.
5. **Sex**: Passenger's sex (male or female).
6. **Age**: Passenger's age in years.
7. **SibSp**: Number of siblings/spouses aboard the Titanic.
8. **Parch**: Number of parents/children aboard the Titanic.
9. **Ticket**: Ticket number.
10. **Fare**: Passenger fare (price of the ticket).
11. **Cabin**: Cabin number where the passenger stayed.
12. **Embarked**: Port of embarkation (C = Cherbourg, Q = Queenstown, S = Southampton).

Explanation:

1. **Figure Layout:**
    - `plt.subplots(2, 3, figsize=(18, 12))` creates a grid of 2 rows and 3 columns for plotting.
2. **Plot 1: Bar Plot of Survival Counts:**
    - Uses `axes[0, 0].bar()` to plot survival counts.
    - Colors bars differently for survival (green) and not survived (red).
3. **Plot 2: Age Distribution of Passengers:**
    - Uses `sns.histplot()` to plot the distribution of passenger ages.
    - `bins=20` divides the age range into 20 bins, `kde=True` adds a kernel density estimate.

4. **Plot 3: Survival Rate by Passenger Class:**

    - Uses `sns.barplot()` to show the survival rate for each passenger class.
    - `ci=None` removes confidence intervals to focus on survival rate.

5. **Plot 4: Scatter Plot of Fare vs Age with Survival:**

    - Uses `sns.scatterplot()` to plot fare against age, coloring points by survival status (`hue='Survived'`).
    - Adds a legend to distinguish between survived (Yes) and did not survive (No).

6. **Plot 5: Count Plot of Passenger Class:**

    - Uses `sns.countplot()` to show the count of passengers in each class.

4. **Plot 3: Survival Rate by Passenger Class:**

    - Uses `sns.barplot()` to show the survival rate for each passenger class.
    - `ci=None` removes confidence intervals to focus on survival rate.

5. **Plot 4: Scatter Plot of Fare vs Age with Survival:**

    - Uses `sns.scatterplot()` to plot fare against age, coloring points by survival status (`hue='Survived'`).
    - Adds a legend to distinguish between survived (Yes) and did not survive (No).

6. **Plot 5: Count Plot of Passenger Class:**

    - Uses `sns.countplot()` to show the count of passengers in each class.

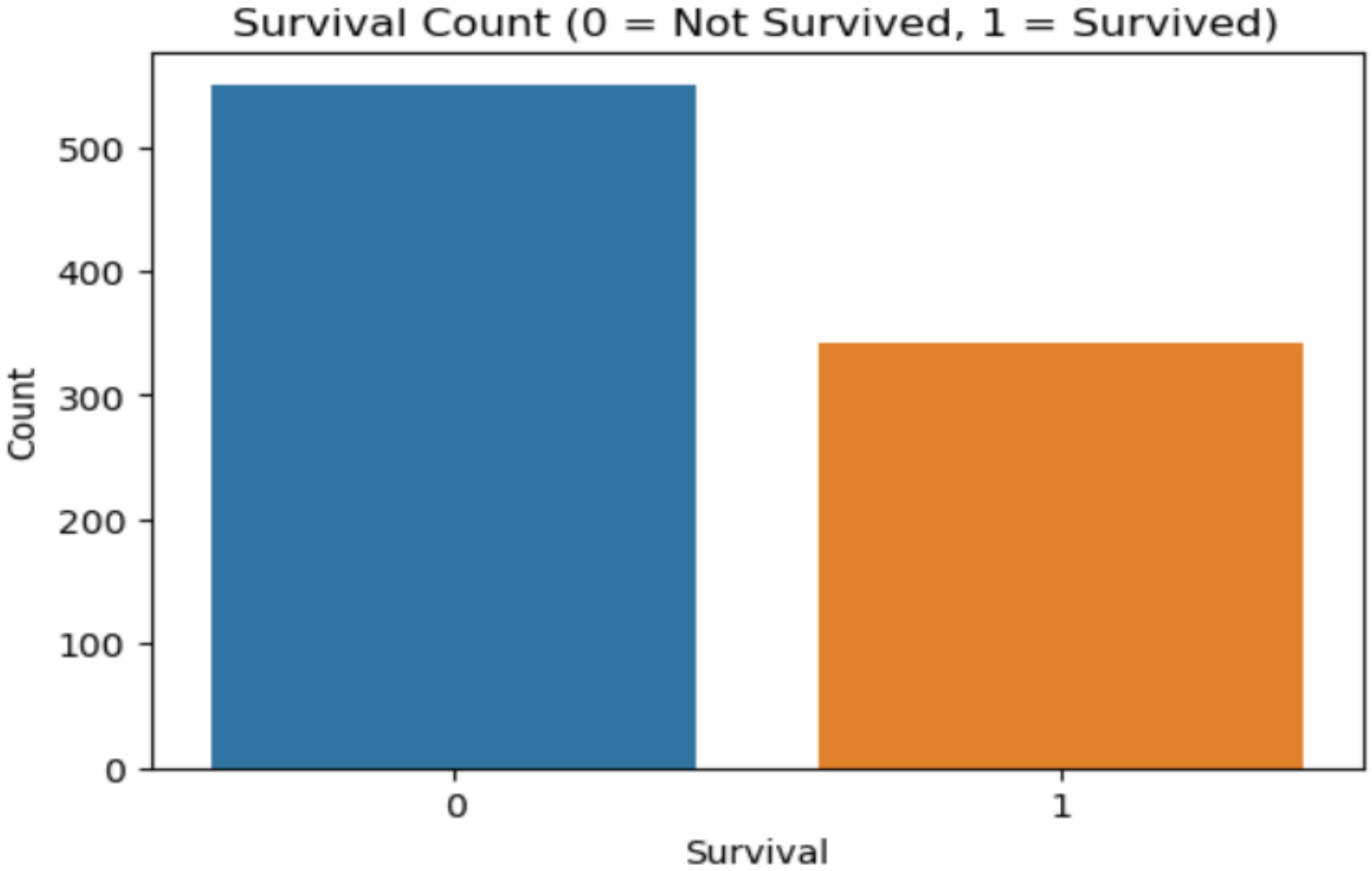
[46]:

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

# Load Titanic dataset using Seaborn
titanic_df = sns.load_dataset('titanic')

# Display the first few rows of the dataset
print(titanic_df.head())
```

```
   survived  pclass     sex   age  sibsp  parch     fare embarked   class  \
0         0       3    male  22.0      1      0   7.2500        S   Third
1         1       1  female  38.0      1      0  71.2833        C   First
2         1       3  female  26.0      0      0   7.9250        S   Third
3         1       1  female  35.0      1      0  53.1000        S   First
4         0       3    male  35.0      0      0   8.0500        S   Third

     who  adult_male deck  embark_town alive  alone
0    man        True  NaN  Southampton    no  False
1  woman       False    C    Cherbourg   yes  False
2  woman       False  NaN  Southampton   yes   True
3  woman       False    C  Southampton   yes  False
4    man        True  NaN  Southampton    no   True
```

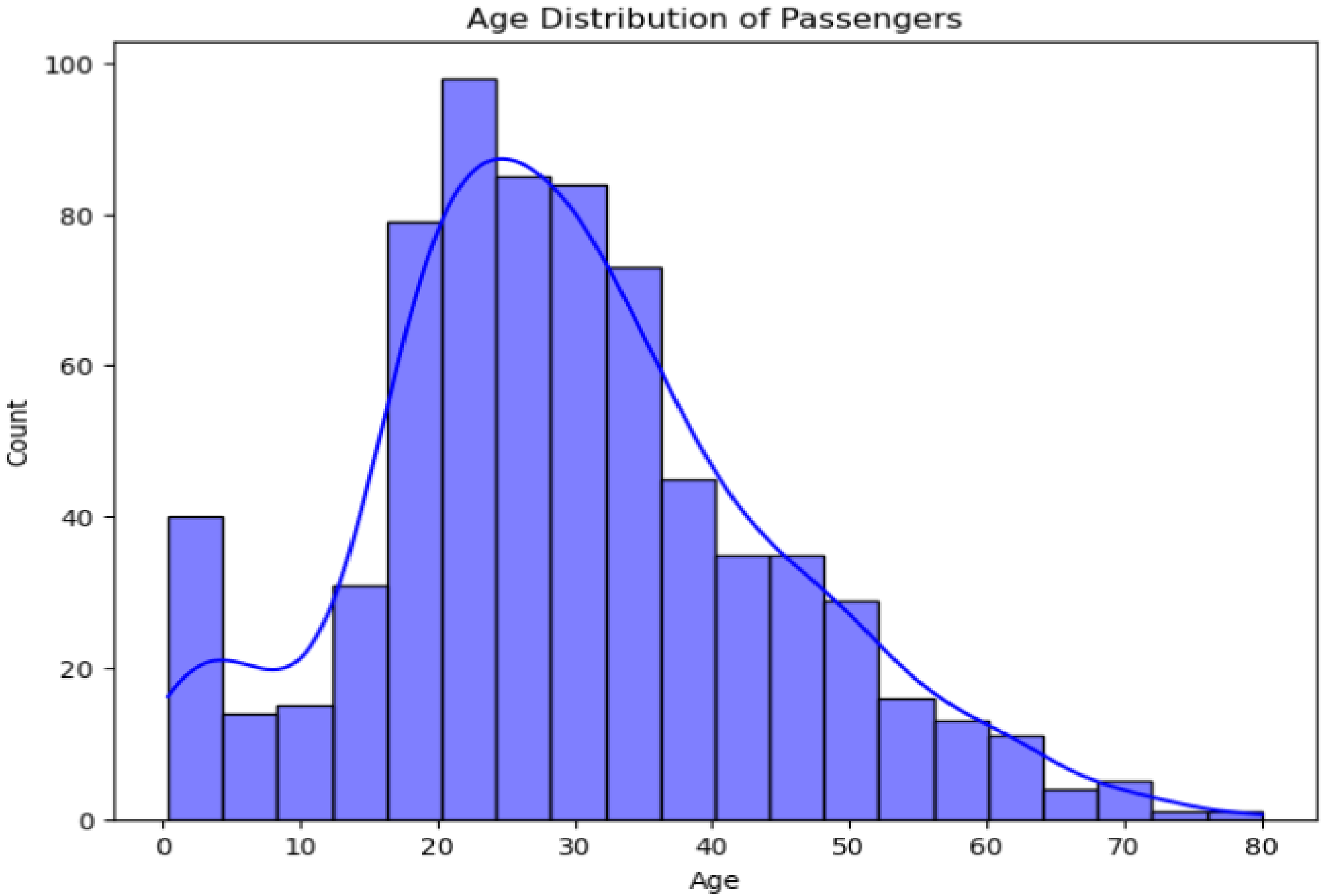
[47]:

```
# Example 1: Bar plot of survival counts
plt.figure(figsize=(6, 4))
sns.countplot(x='survived', data=titanic_df)
plt.title('Survival Count (0 = Not Survived, 1 = Survived)')
plt.xlabel('Survival')
plt.ylabel('Count')
plt.show()
```


Survival Count (0 = Not Survived, 1 = Survived)
Count
500
400
300
200
100
0
0
1
Survival

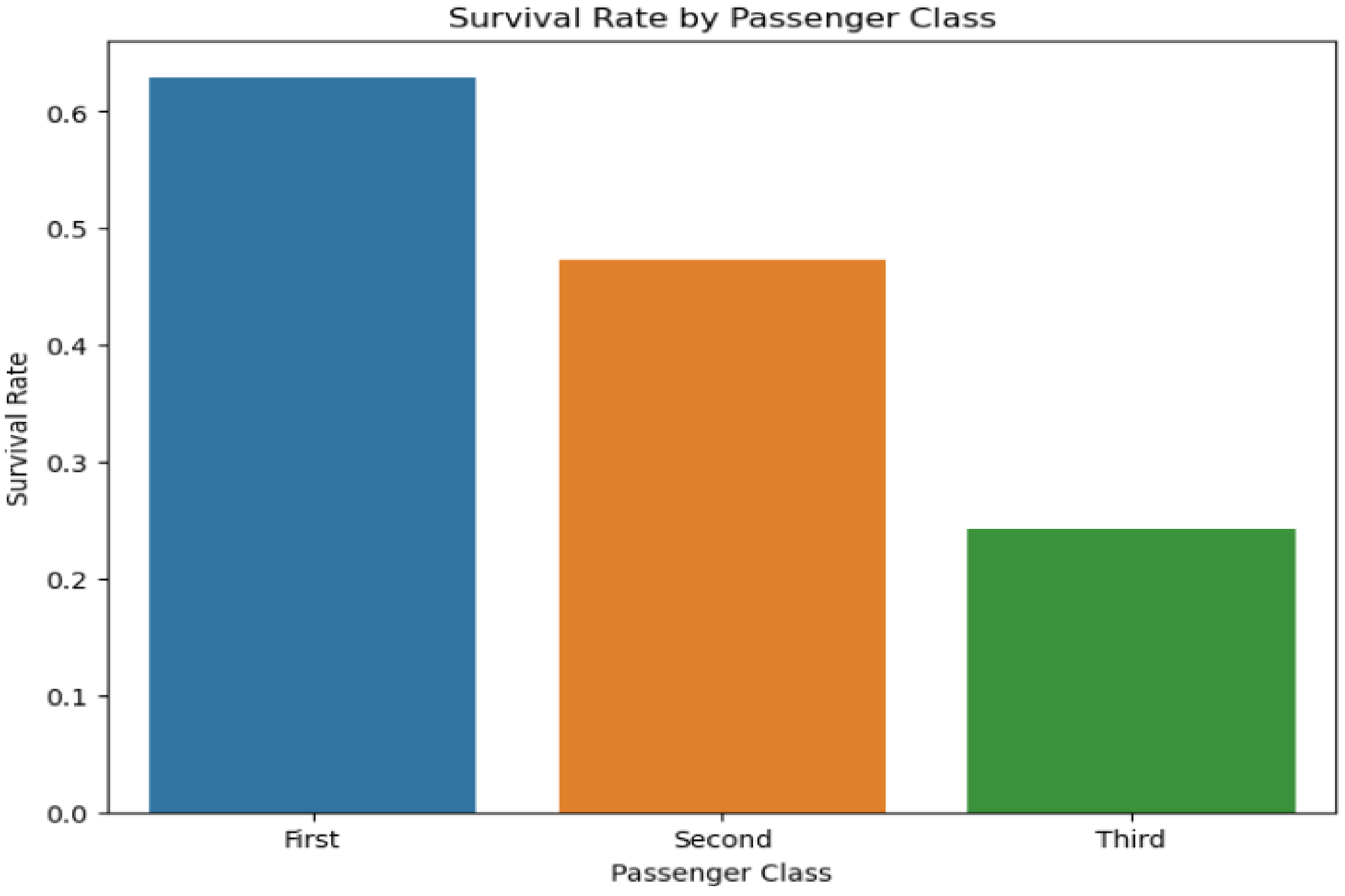
[48]:

```
# Example 2: Age distribution of passengers
plt.figure(figsize=(8, 6))
sns.histplot(titanic_df['age'].dropna(), bins=20, kde=True, color='blue')
plt.title('Age Distribution of Passengers')
plt.xlabel('Age')
plt.ylabel('Count')
plt.show()
```


Age Distribution of Passengers
Count
Age
0
20
40
60
80
100
0
10
20
30
40
50
60
70
80

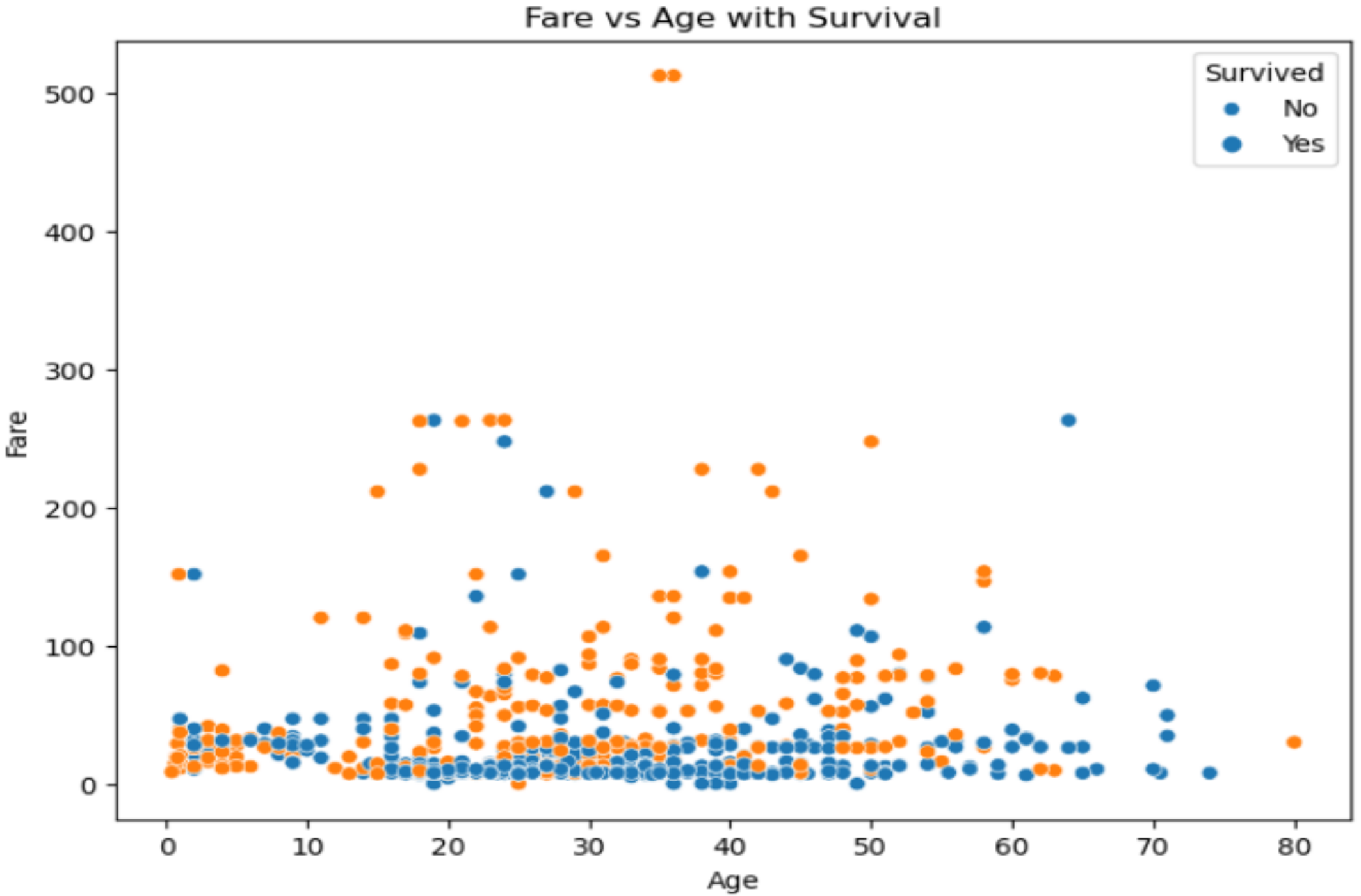
[49]:

```
# Example 3: Survival rate by passenger class
plt.figure(figsize=(8, 6))
sns.barplot(x='class', y='survived', data=titanic_df, ci=None)
plt.title('Survival Rate by Passenger Class')
plt.xlabel('Passenger Class')
plt.ylabel('Survival Rate')
plt.show()
```


Survival Rate by Passenger Class
Survival Rate
0.0
0.1
0.2
0.3
0.4
0.5
0.6
First
Second
Third
Passenger Class

[50]:

```
# Example 4: Scatter plot of Fare vs Age
plt.figure(figsize=(8, 6))
sns.scatterplot(x='age', y='fare', data=titanic_df, hue='survived')
plt.title('Fare vs Age with Survival')
plt.xlabel('Age')
plt.ylabel('Fare')
plt.legend(title='Survived', loc='upper right', labels=['No', 'Yes'])
plt.show()
```


Fare vs Age with Survival
Survived
No
Yes
Fare
Age
0
100
200
300
400
500
0
10
20
30
40
50
60
70
80

Seaborn

- **Seaborn** is a Python data visualization library built on top of Matplotlib.
- It provides a high-level interface for drawing attractive and informative statistical graphics.
- Seaborn simplifies the creation of complex visualizations and integrates well with Pandas DataFrames, making it a powerful tool for data analysis and visualization.

Getting Started with Seaborn

1. **Installation**

To install Seaborn, use pip:

```bash
pip install seaborn
```

- The tips dataset is a built-in example dataset provided by Seaborn.
- It contains data from a restaurant, specifically focusing on the tips received by waitstaff.
- This dataset is often used for demonstrating and practicing data visualization techniques.

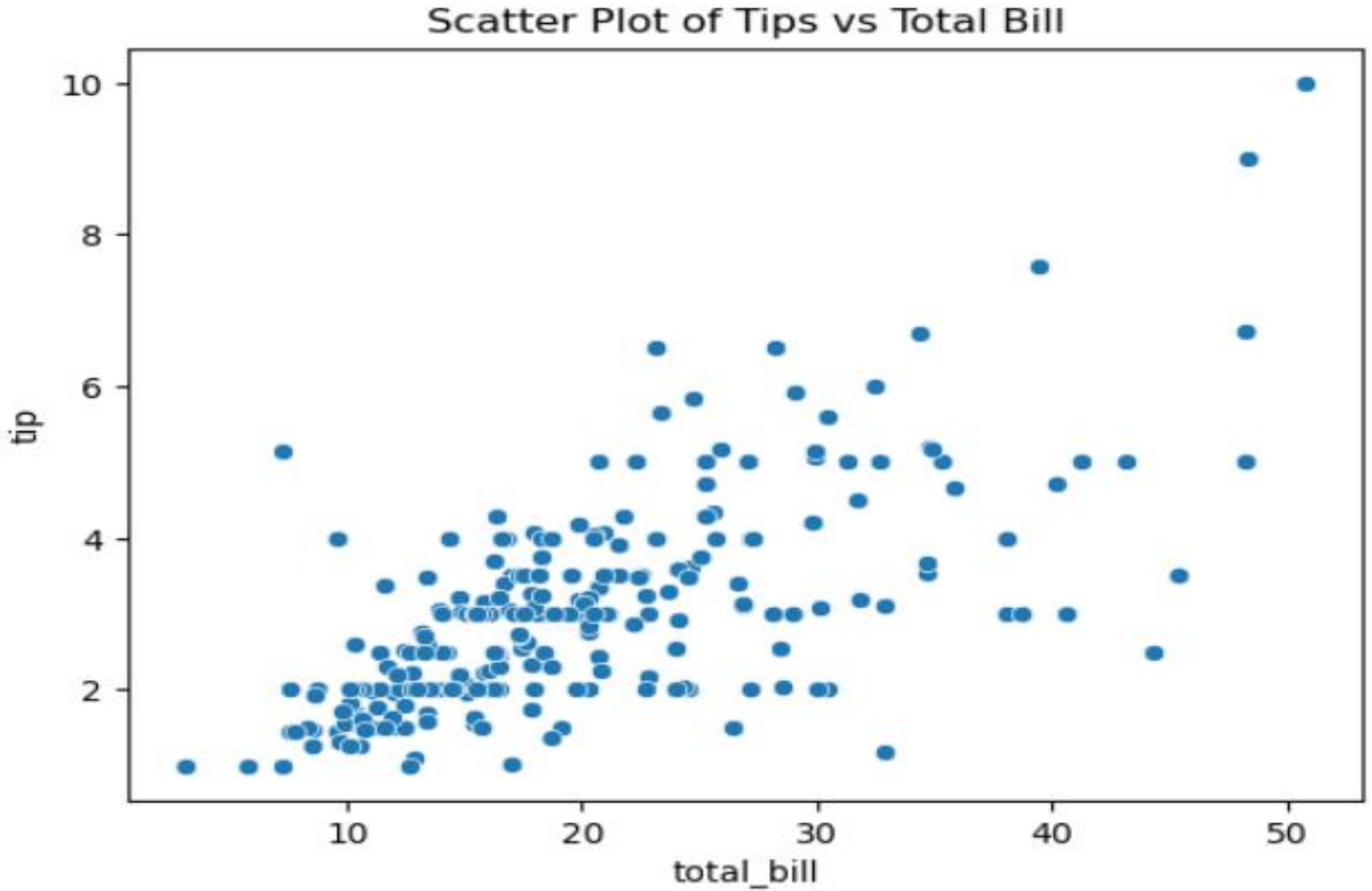
Structure of the `tips` Dataset

The `tips` dataset includes the following columns:

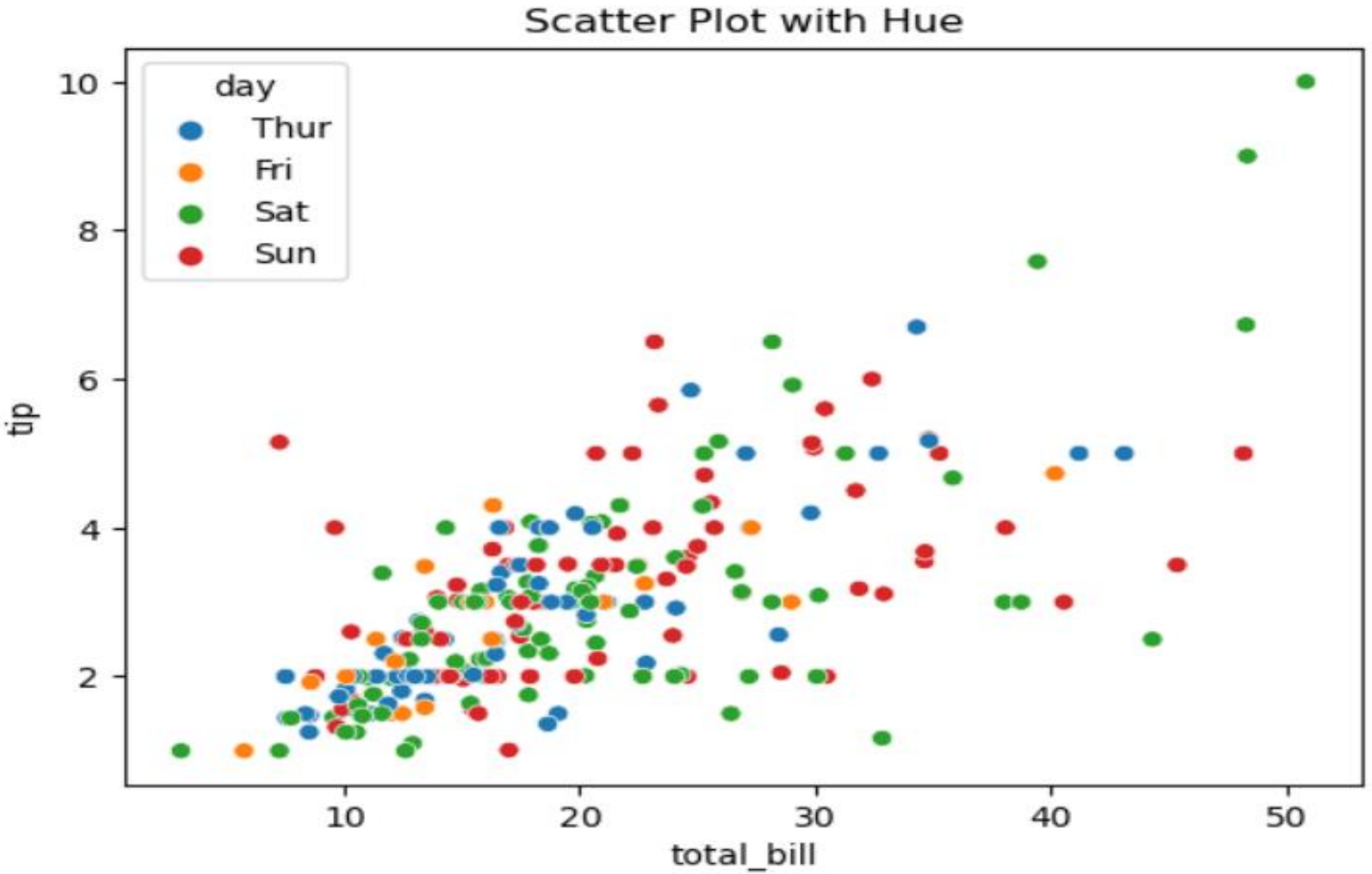
1. **total_bill**: The total amount of the bill in dollars.
2. **tip**: The amount of the tip in dollars.
3. **sex**: The gender of the person who paid the bill (Male or Female).
4. **smoker**: Indicates whether the person was a smoker (Yes or No).
5. **day**: The day of the week when the bill was paid (Thur, Fri, Sat, Sun).
6. **time**: The time of day when the bill was paid (Lunch or Dinner).
7. **size**: The size of the party (number of people).

[1]:

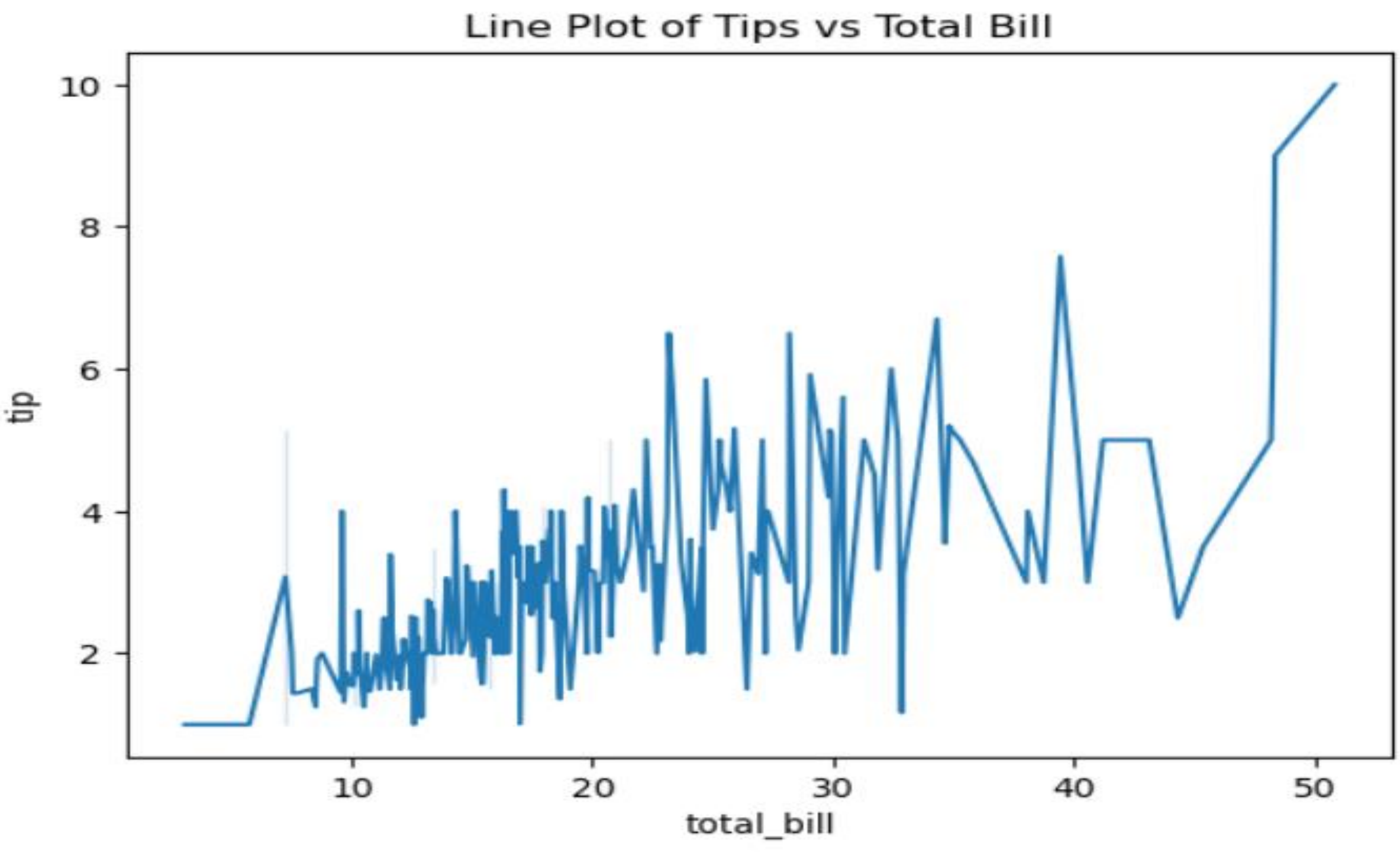
```
import seaborn as sns
import matplotlib.pyplot as plt

# Load an example dataset
data = sns.load_dataset('tips')

# Create a scatter plot
sns.scatterplot(x='total_bill', y='tip', data=data)

# Add a title
plt.title('Scatter Plot of Tips vs Total Bill')

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



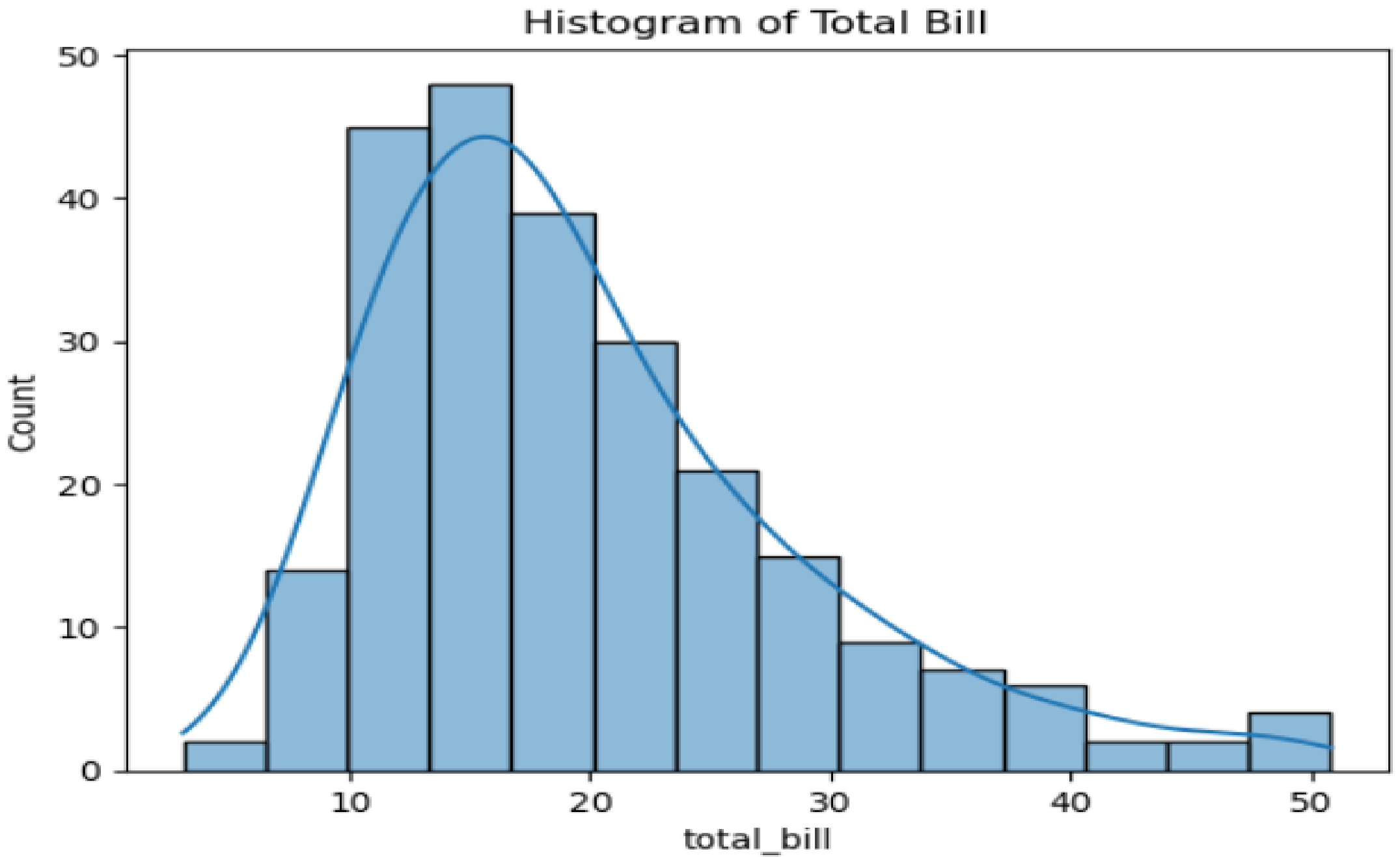
[2]:
```
sns.scatterplot(x='total_bill', y='tip', hue='day', data=data)  # 'hue' adds color coding
plt.title('Scatter Plot with Hue')
plt.show()
```


Scatter Plot with Hue
day
Thur
Fri
Sat
Sun
tip
total_bill
2
4
6
8
10
10
20
30
40
50

[3]:
```
sns.lineplot(x='total_bill', y='tip', data=data, estimator='mean')  # 'estimator' can be 'mean', 'median', etc.
plt.title('Line Plot of Tips vs Total Bill')
plt.show()
```

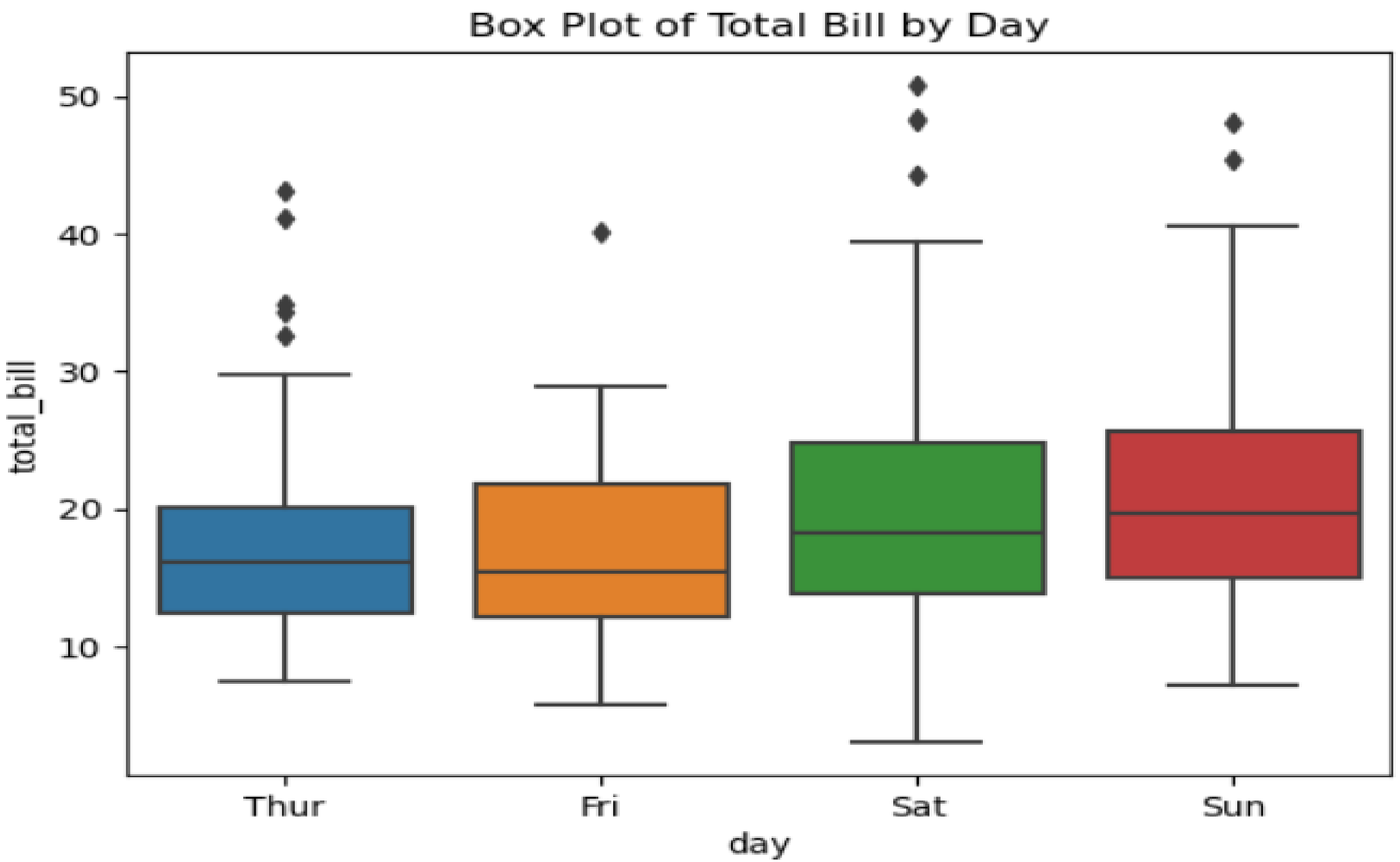

Line Plot of Tips vs Total Bill
tip
total_bill
10
8
6
4
2
10
20
30
40
50

[4]:
```
sns.histplot(data['total_bill'], kde=True)  # 'kde' adds a Kernel Density Estimate
plt.title('Histogram of Total Bill')
plt.show()
```



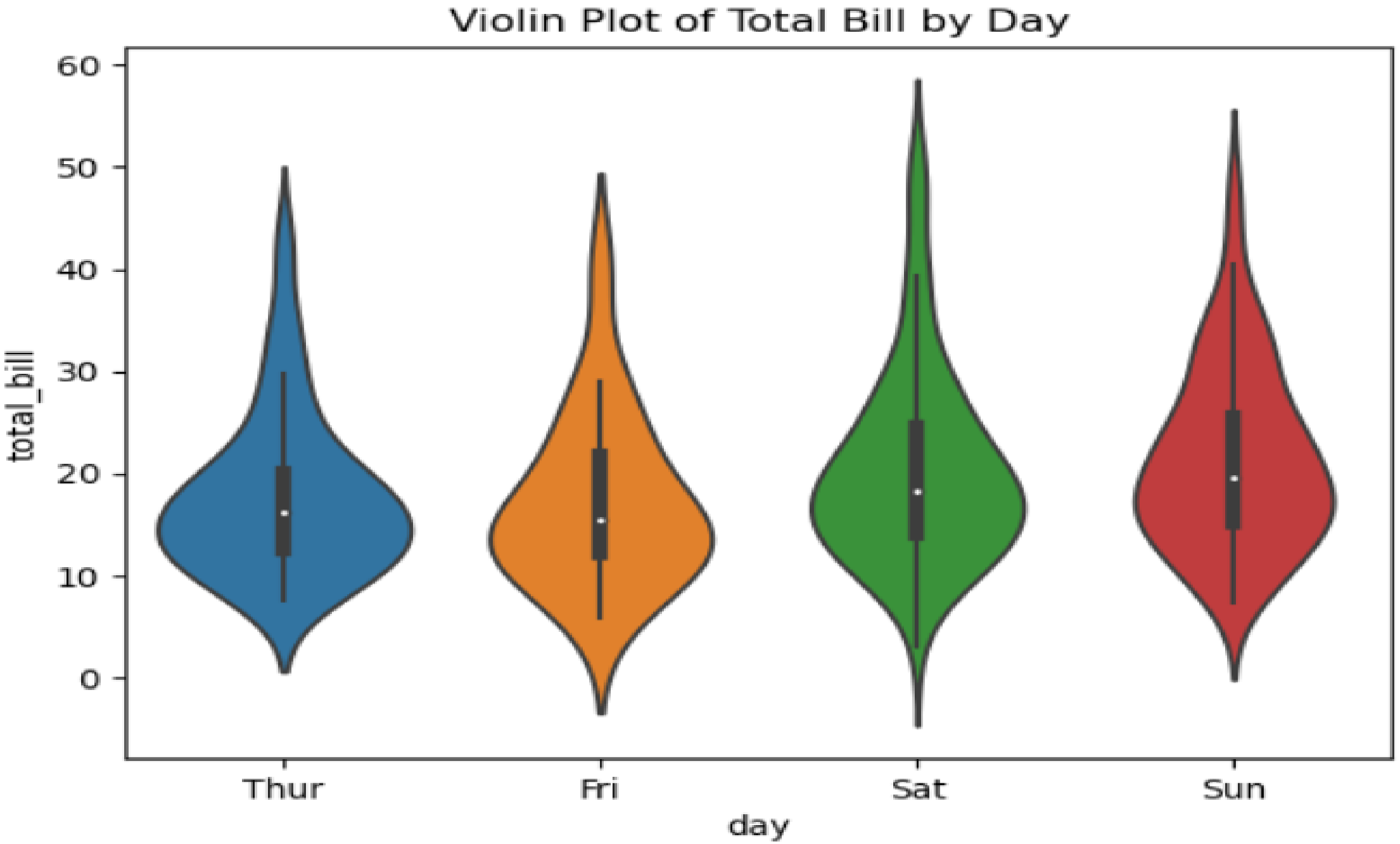
[5]:

```
sns.boxplot(x='day', y='total_bill', data=data)
plt.title('Box Plot of Total Bill by Day')
plt.show()
```

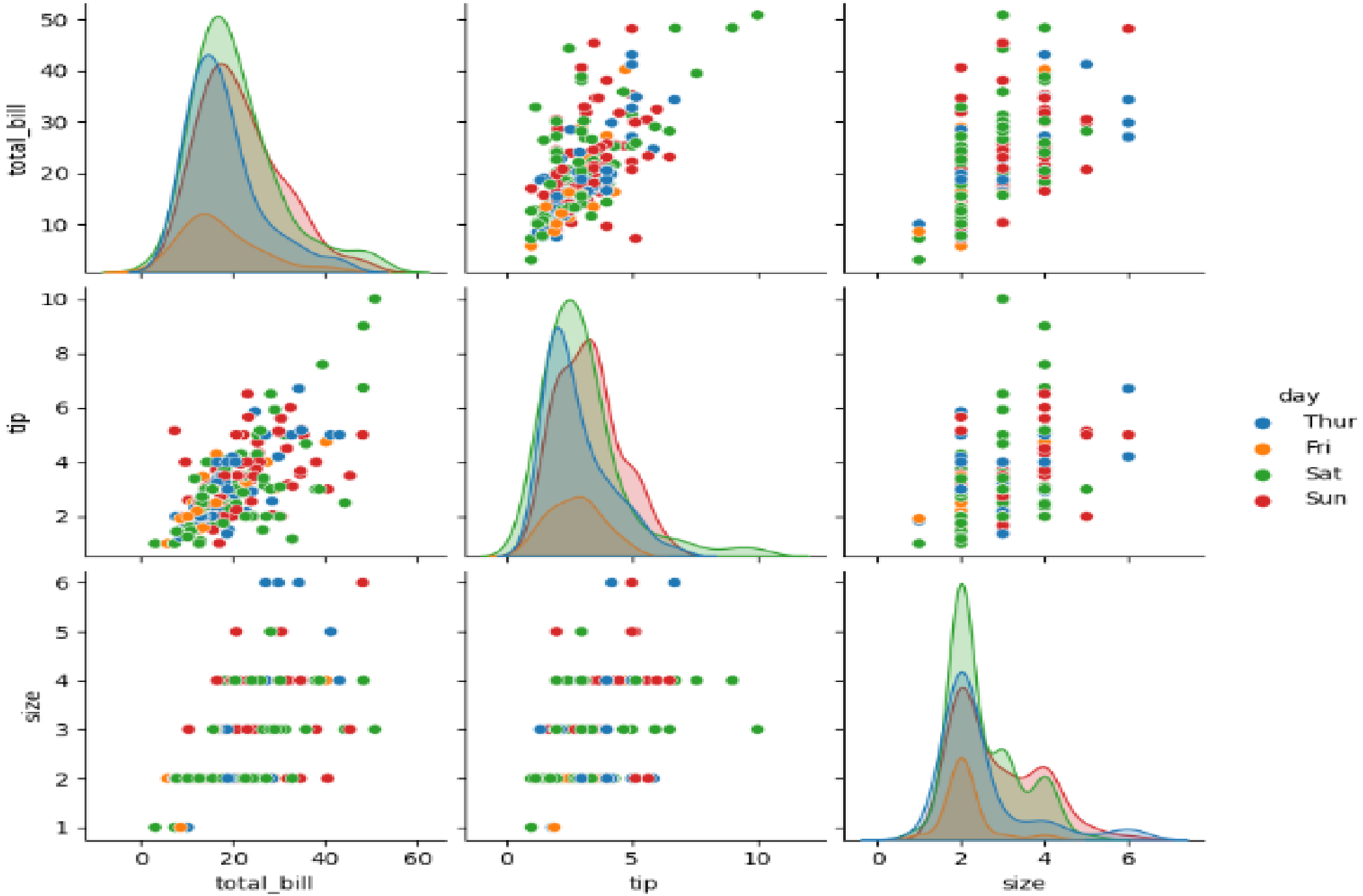

Box Plot of Total Bill by Day
total_bill
50
40
30
20
10
Thur
Fri
Sat
Sun
day

[6]:

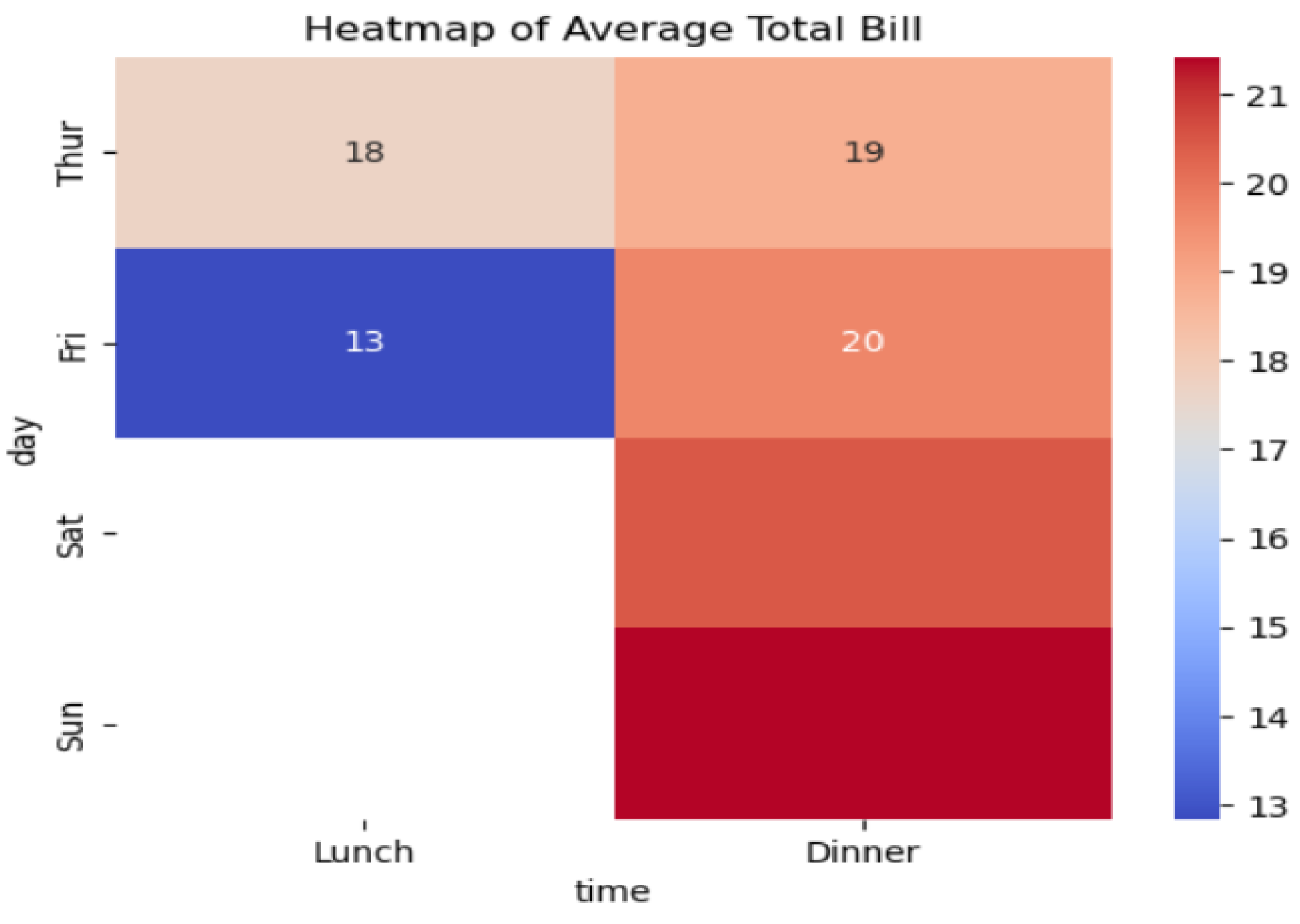
```
sns.violinplot(x='day', y='total_bill', data=data)
plt.title('Violin Plot of Total Bill by Day')
plt.show()
```


Violin Plot of Total Bill by Day
total_bill
60
50
40
30
20
10
0
Thur
Fri
Sat
Sun
day

[7]:
```
sns.pairplot(data, hue='day')  # Shows pairwise relationships in a dataset
plt.show()
```

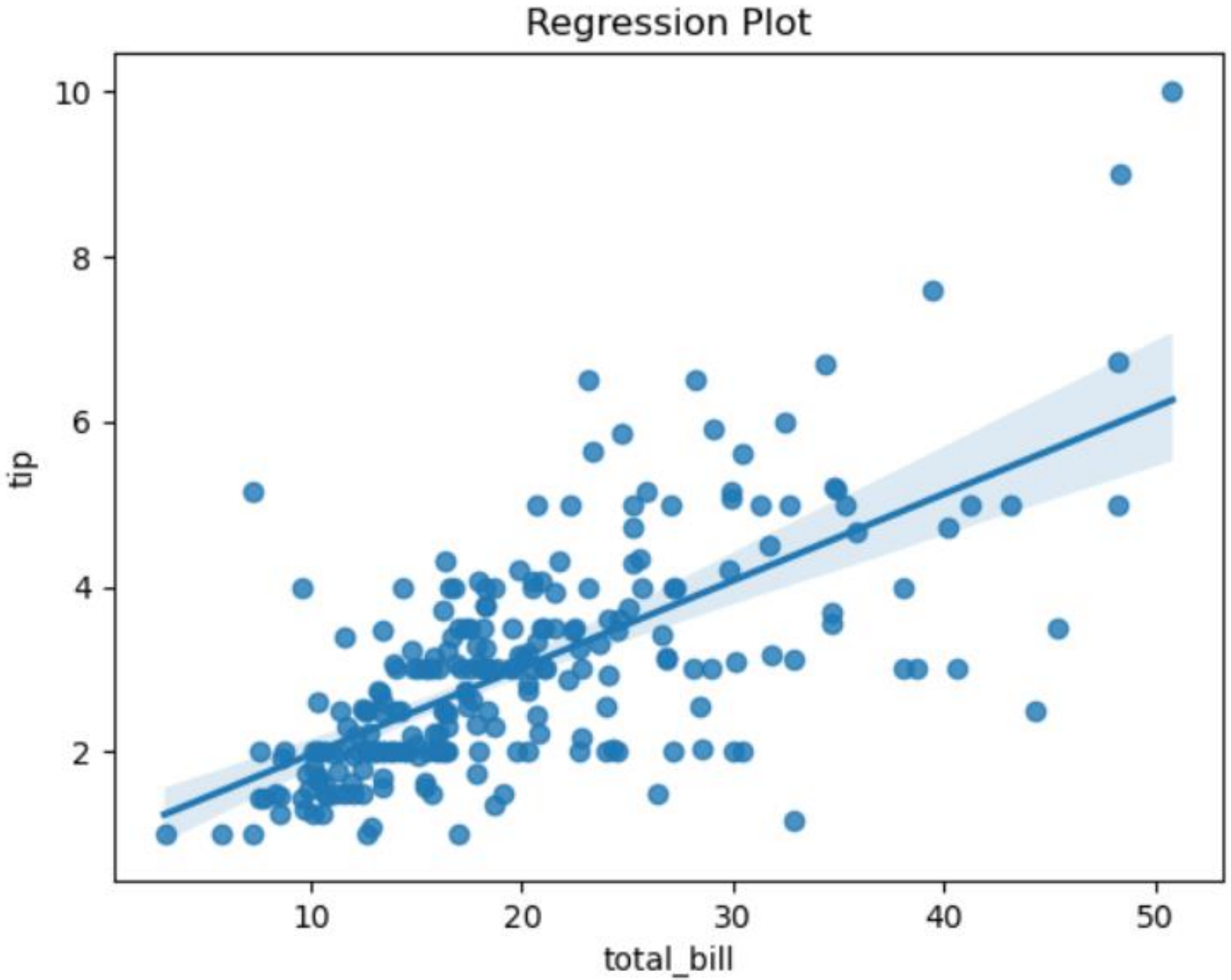

total_bill
tip
size
day
Thur
Fri
Sat
Sun

[8]:
```
# Pivot the data to create a heatmap
pivot_table = data.pivot_table(values='total_bill', index='day', columns='time', aggfunc='mean')

sns.heatmap(pivot_table, annot=True, cmap='coolwarm')  # 'annot' adds the data values to the heatmap
plt.title('Heatmap of Average Total Bill')
plt.show()
```

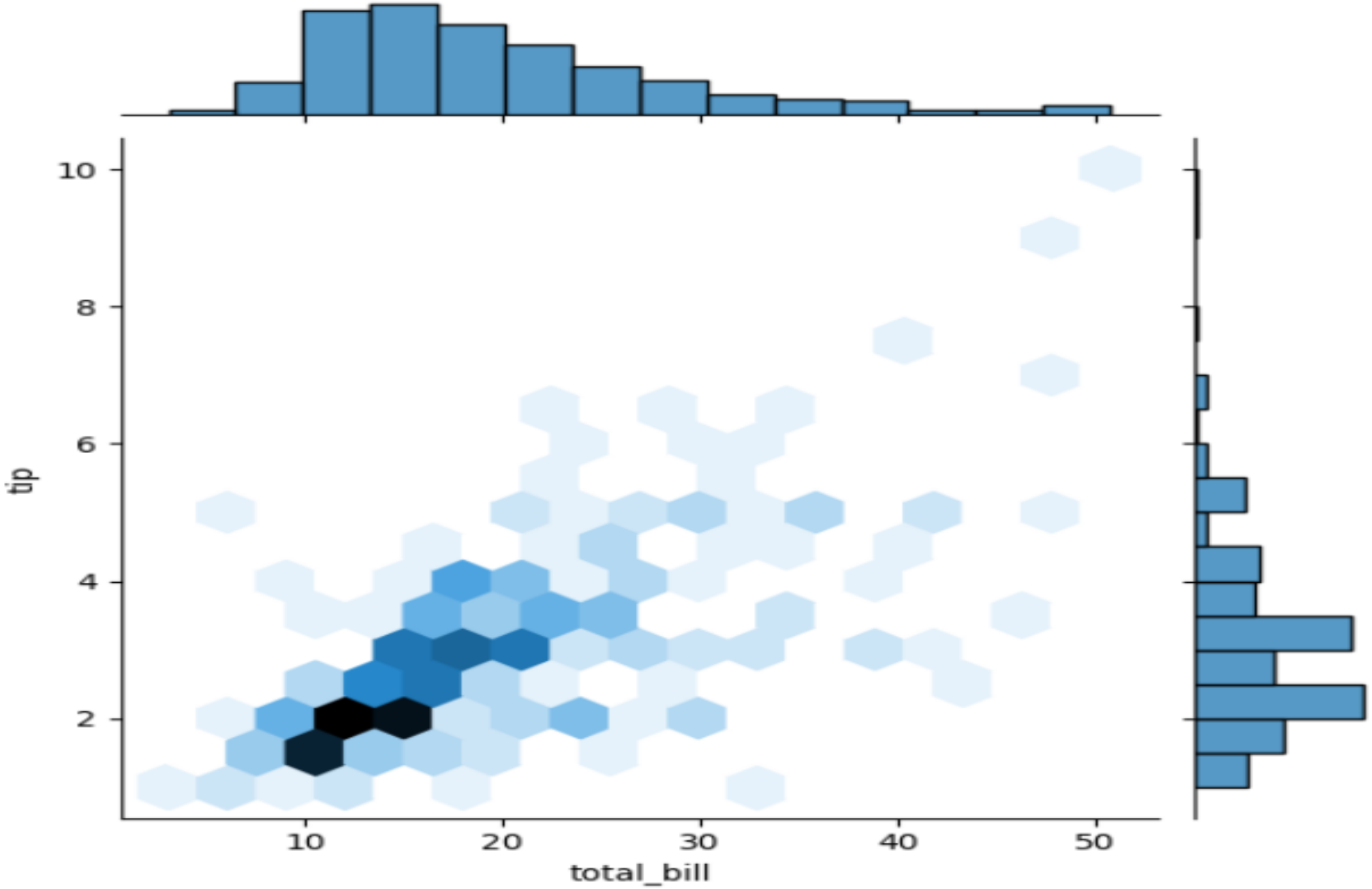

Heatmap of Average Total Bill
Thur
Fri
Sat
Sun
day
18
19
13
20
Lunch
Dinner
time
21
20
19
18
17
16
15
14
13

[9]:

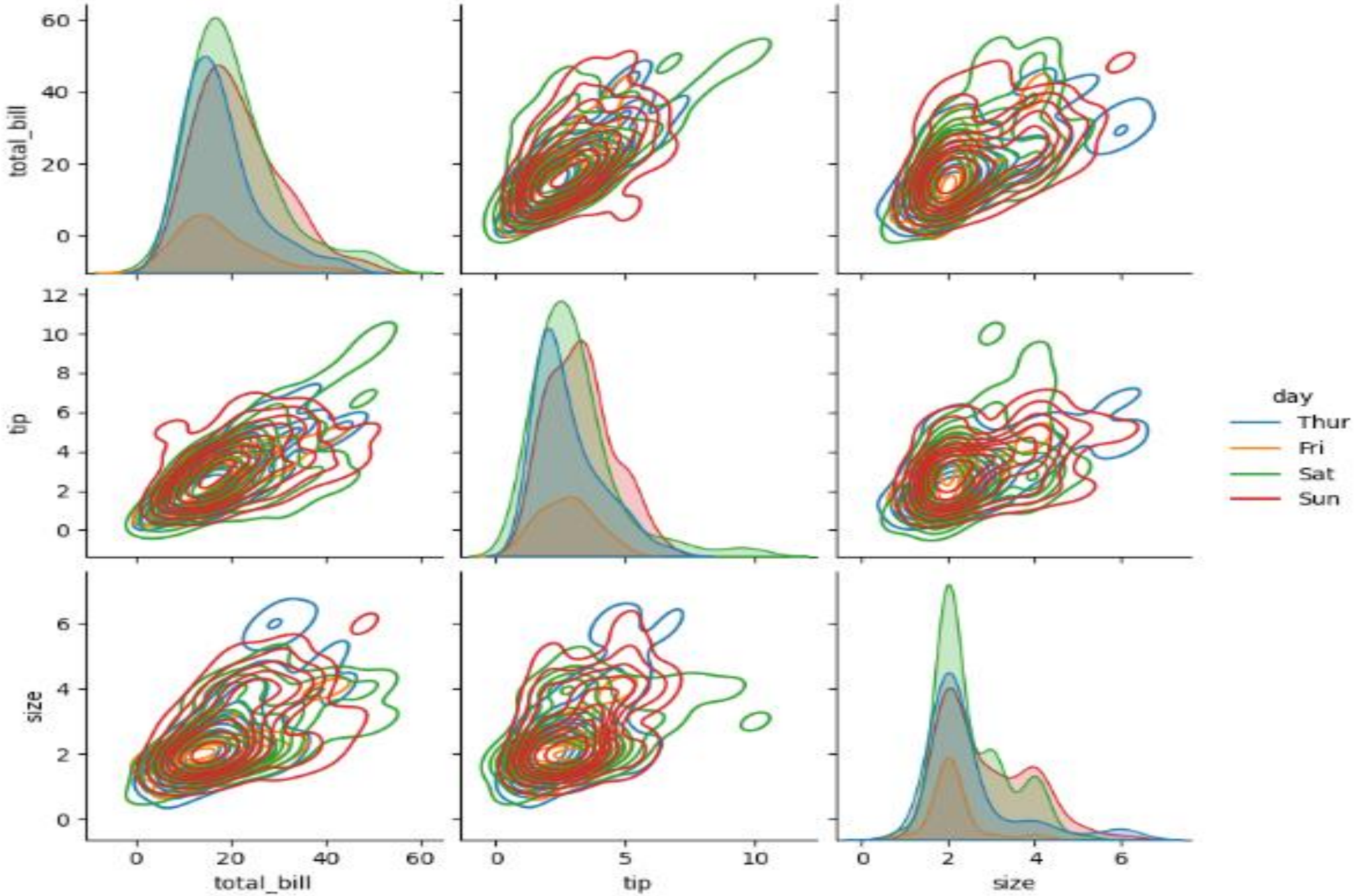
```
sns.regplot(x='total_bill', y='tip', data=data)
plt.title('Regression Plot')
plt.show()
```

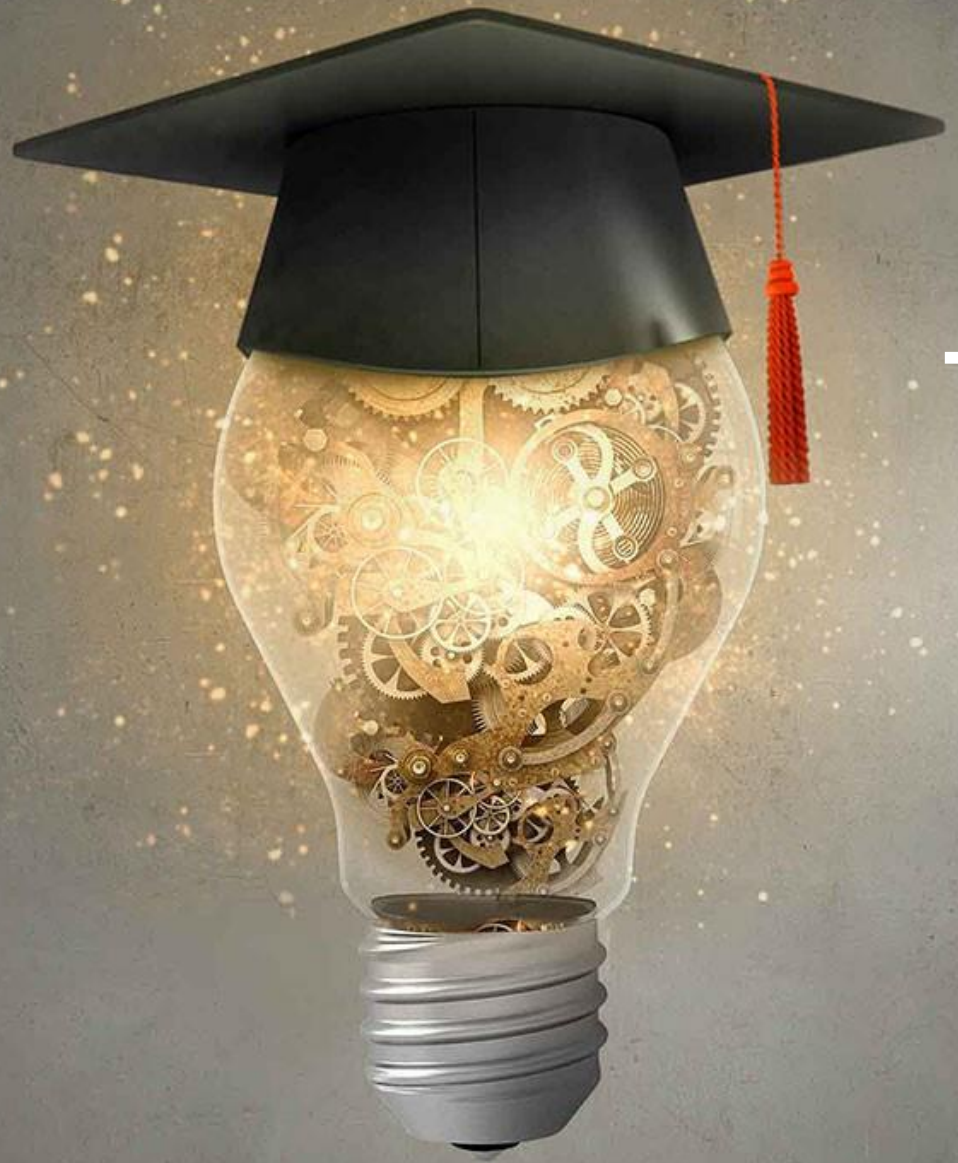

Regression Plot
tip
total_bill
10
8
6
4
2
10
20
30
40
50

[10]:
```
sns.jointplot(x='total_bill', y='tip', data=data, kind='hex')  # Options: 'scatter', 'reg', 'resid', 'kde', 'hex'
plt.show()
```



[11]:
```
sns.pairplot(data, hue='day', kind='kde')  # 'kind' can be 'scatter' or 'kde'
plt.show()
```





THANK YOU

VOLTUS AI ACADEMY