Introduction to Matplotlib

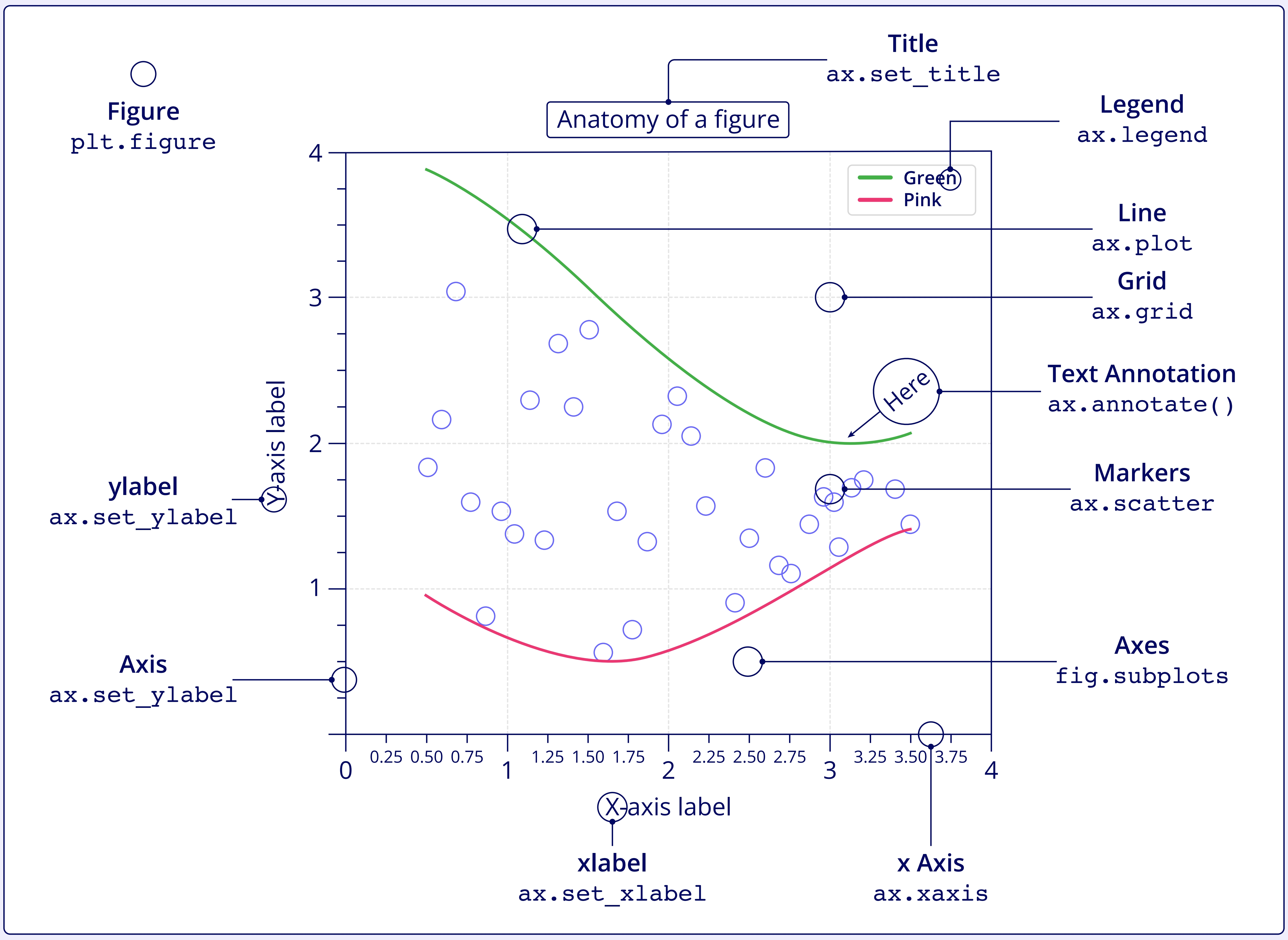
What is Matplotlib?

Matplotlib is a plotting library for the Python programming language and its numerical mathematics extension NumPy. It provides a wide variety of plots and charts for visualizing data.

Why use Matplotlib for Data Visualization?

Used for its flexibility and ease of use, Matplotlib allows users to create high-quality plots with customizable features.

Plotting with Matplotlib



Typical Workflow of Plotting

a

Import necessary libraries

```
import matplotlib.pyplot as plt
import numpy as np # (optional, if we need to work with numerical data)
```

b

Prepare your data

Organize your data in a format that Matplotlib can use for plotting. This often involves using NumPy arrays or Python lists.
For example:

```
# Data preparation: Creating two lists representing x and y values
x = np.array([1, 2, 3, 4, 5])  # Array representing x values
y = np.array([10, 15, 13, 18, 16])  # Array representing y values
```

c

Create a plot(s)

Use Matplotlib functions to create different types of plots, such as line plots, bar plots, scatter plots, histograms, etc.

For example, for a line plot:

- **Using the pyplot interface:**

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

- **Using an object-oriented interface:**

```
fig, ax = plt.subplots() # Create a figure and an axis object
ax.plot(x,y) # Plot data on the axis
```

***Note:**

`fig:` This represents the entire figure object, including the entire graphical representation, such as axes, labels, legends, etc. It's like the canvas on which our plots are drawn.

`ax:` This represents an individual subplot or axis within the larger figure. Think of it as a container for a specific plot or chart. We can have multiple axes within a single figure, each representing a separate plot.

d

Customize the plot

Add labels, titles, legends, grid lines, colors, and other customizations to make the plot more informative and visually appealing.

For example:

```
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.title('Simple Line Plot')
plt.grid(True)
```

Or

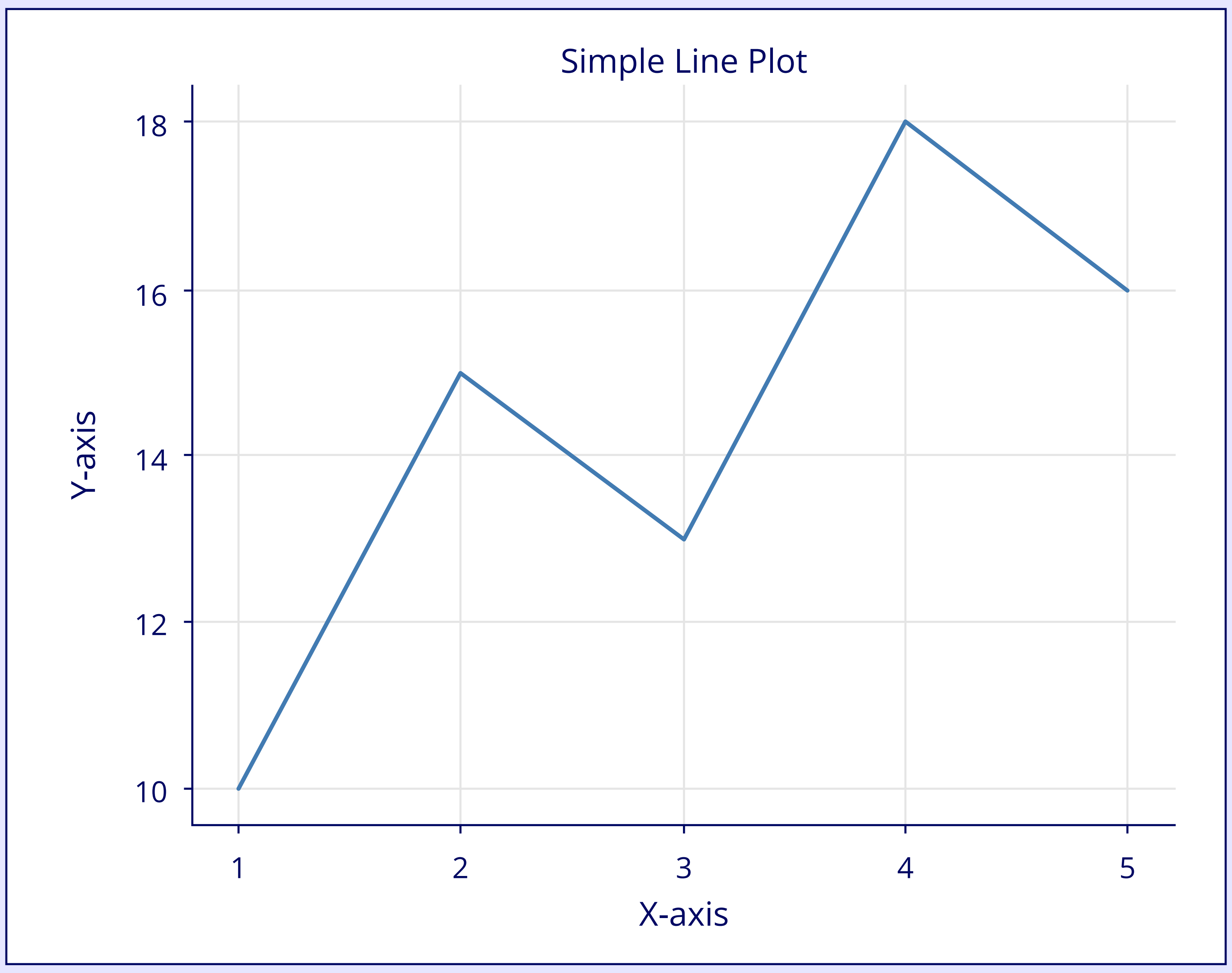
```
ax.set_xlabel("X-axis")
ax.set_ylabel("Y-axis")
ax.set_title("Simple Line Plot")
ax.grid(True)
```

e

Save and display the plot

Use **`plt.show()`** to display the plot on your screen. You can also save the plot to a file using **`plt.savefig()`**.

```
plt.show()
```



f

Interact with the plot (Optional)

Matplotlib provides interactive features for exploring the plot, such as zooming, panning, and saving specific parts of the plot.

Basic Plotting

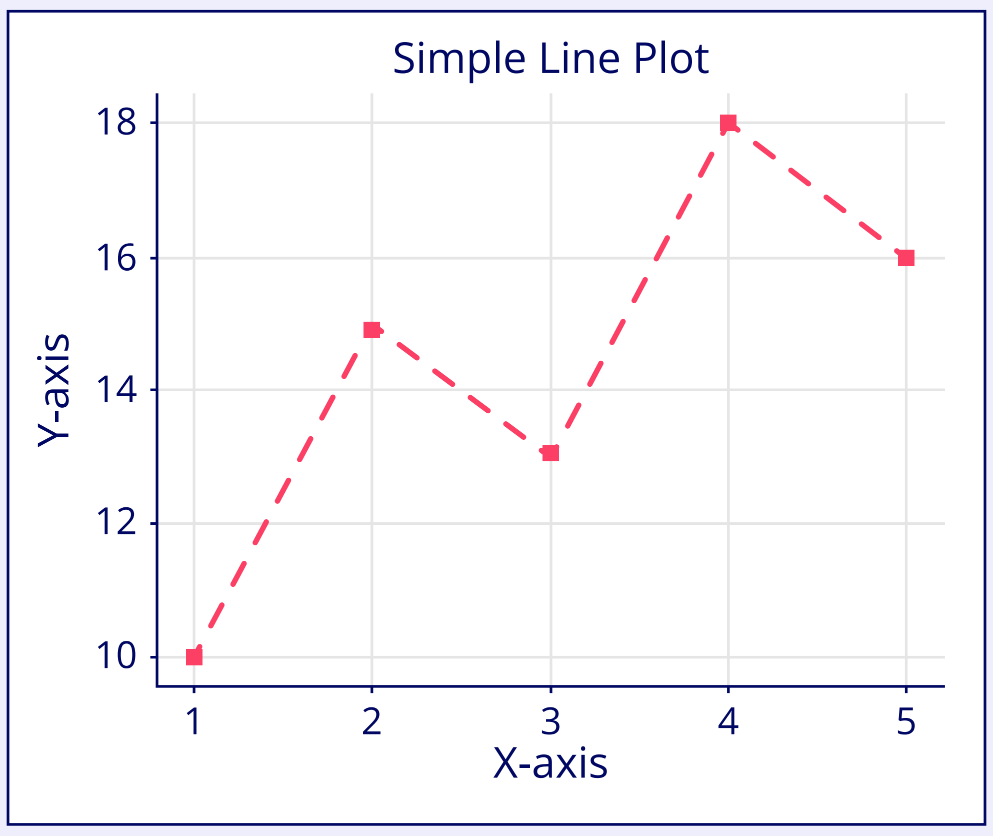
Line Plot

- Creating a simple line plot

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

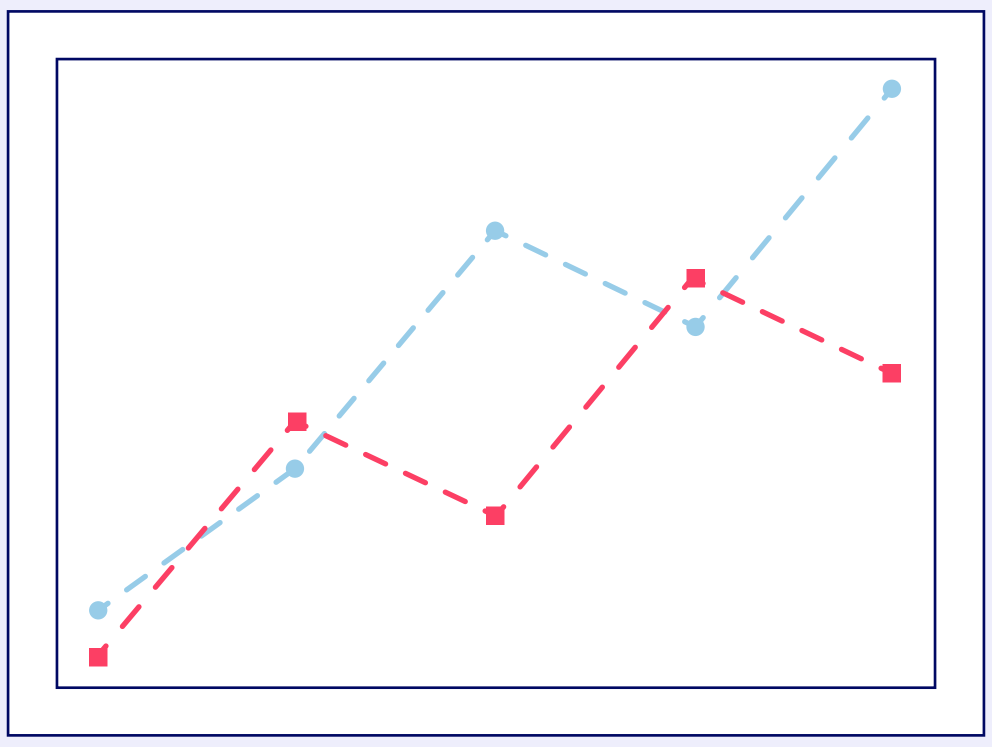
- Customizing line styles, marker styles, and colors
  For example:

```
#Setting marker 's' uses a square marker for the plot points
plt.plot(x, y, linestyle='--', marker ='s', color='red')
```



For example:

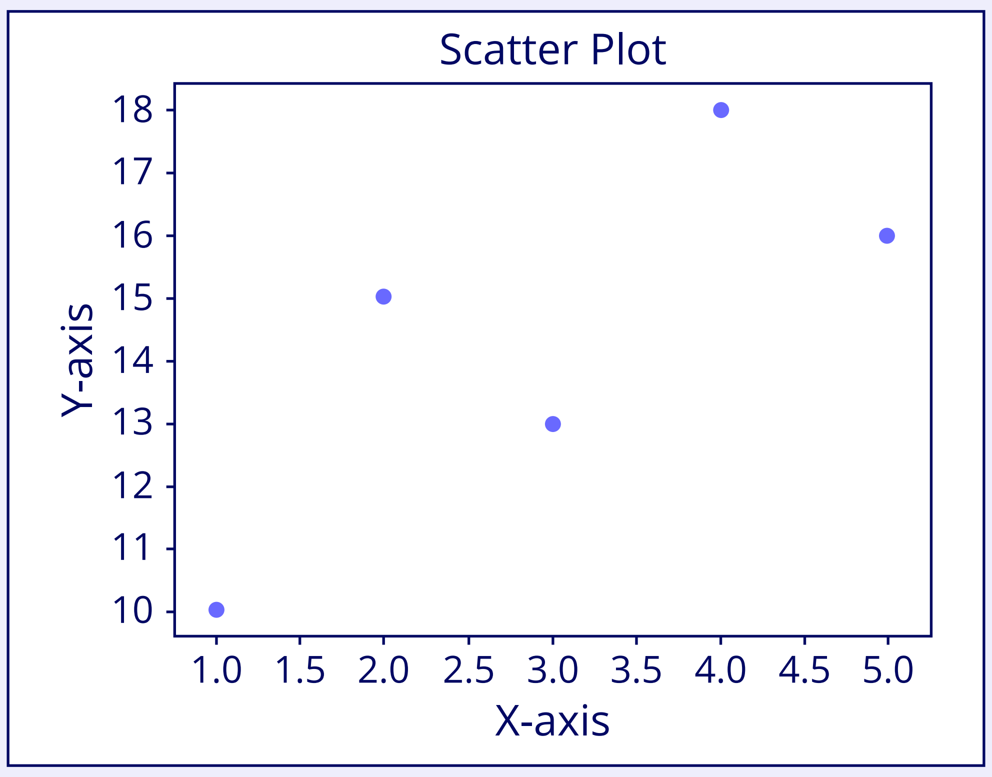
```
#Setting marker 's' uses a square marker and ‘o’ uses a circle marker for the plot points
plt.plot(x,y1, linestyle='--', marker='s', color='red')
plt.plot(x,y2, linestyle='--', marker='o', color='skyblue')
```



Scatter Plot

- Creating a scatter plot

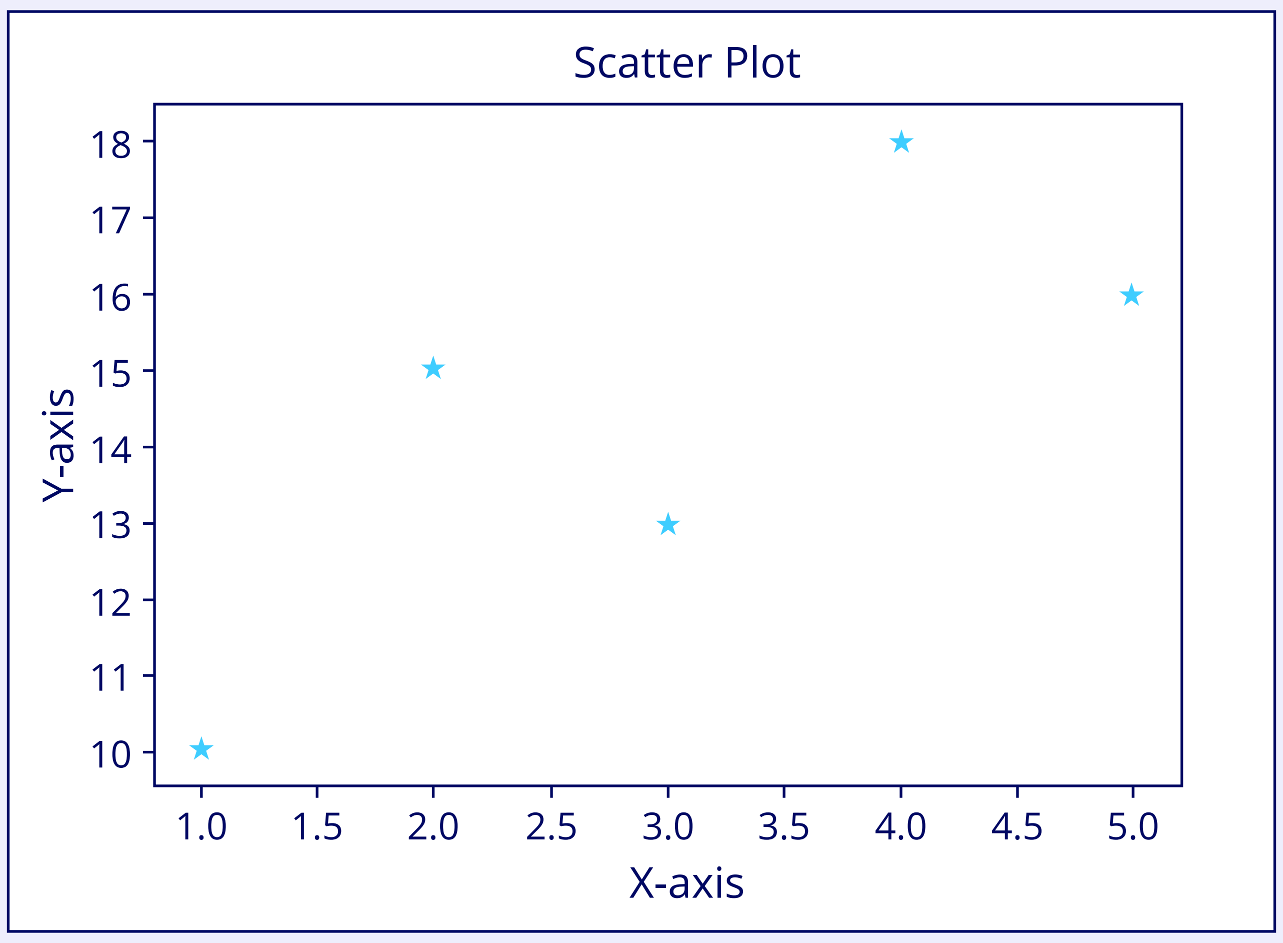
```
plt.scatter(x, y)
```



- Customizing marker styles and colors
  - (s)ize, (c)olor, cmap, marker, area, and alpha

  For example:

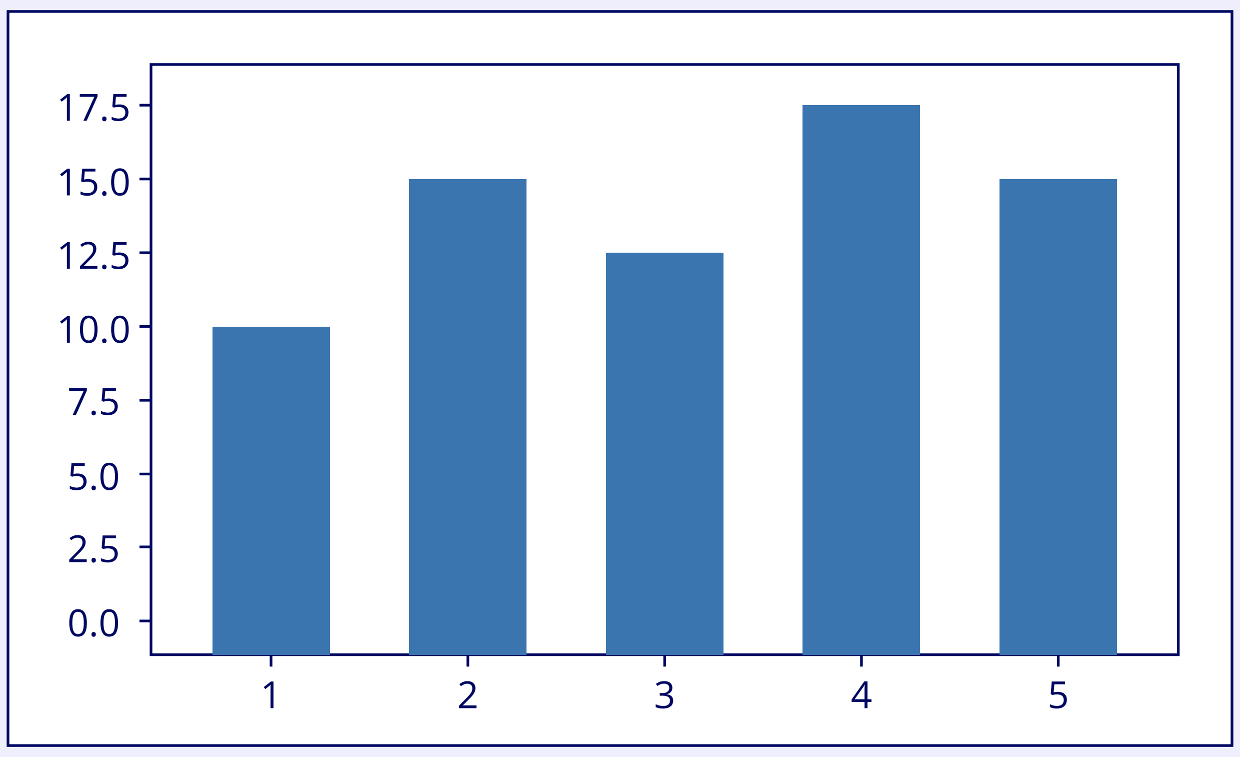
```
plt.scatter(x, y, marker='*', color='blue')
```



Bar Plot

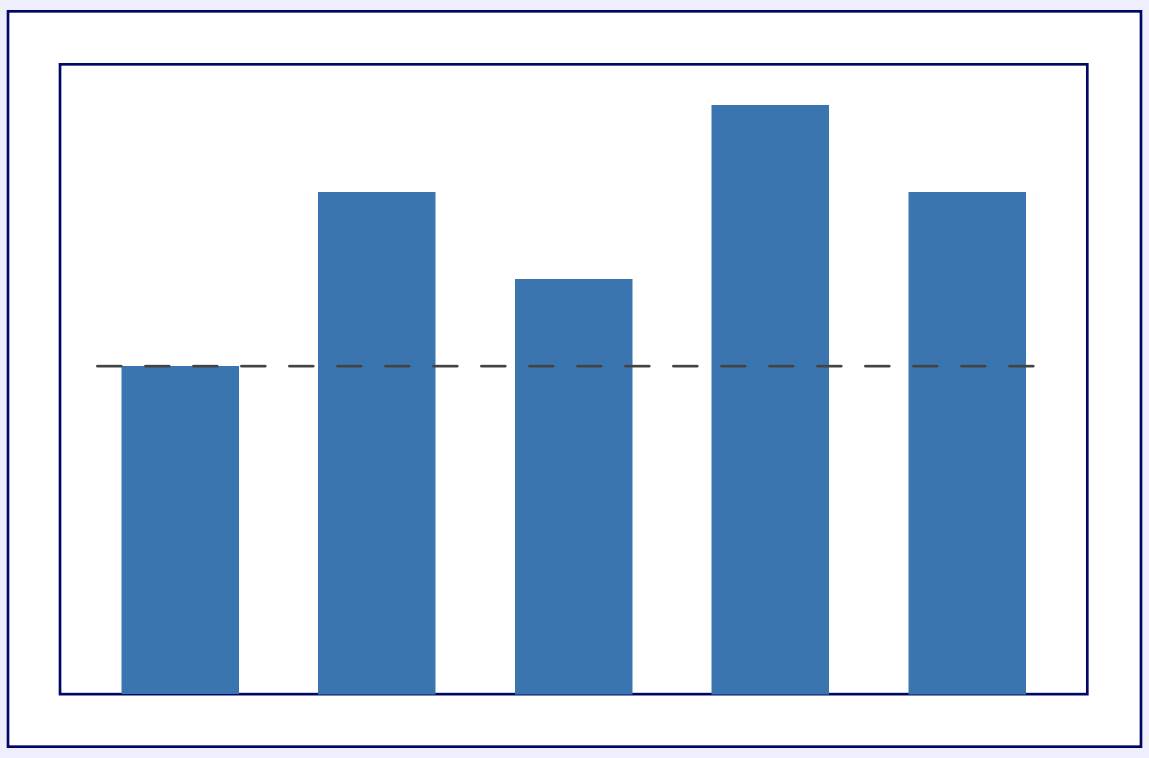
- Creating a bar plot

```
plt.bar(x, height)
```



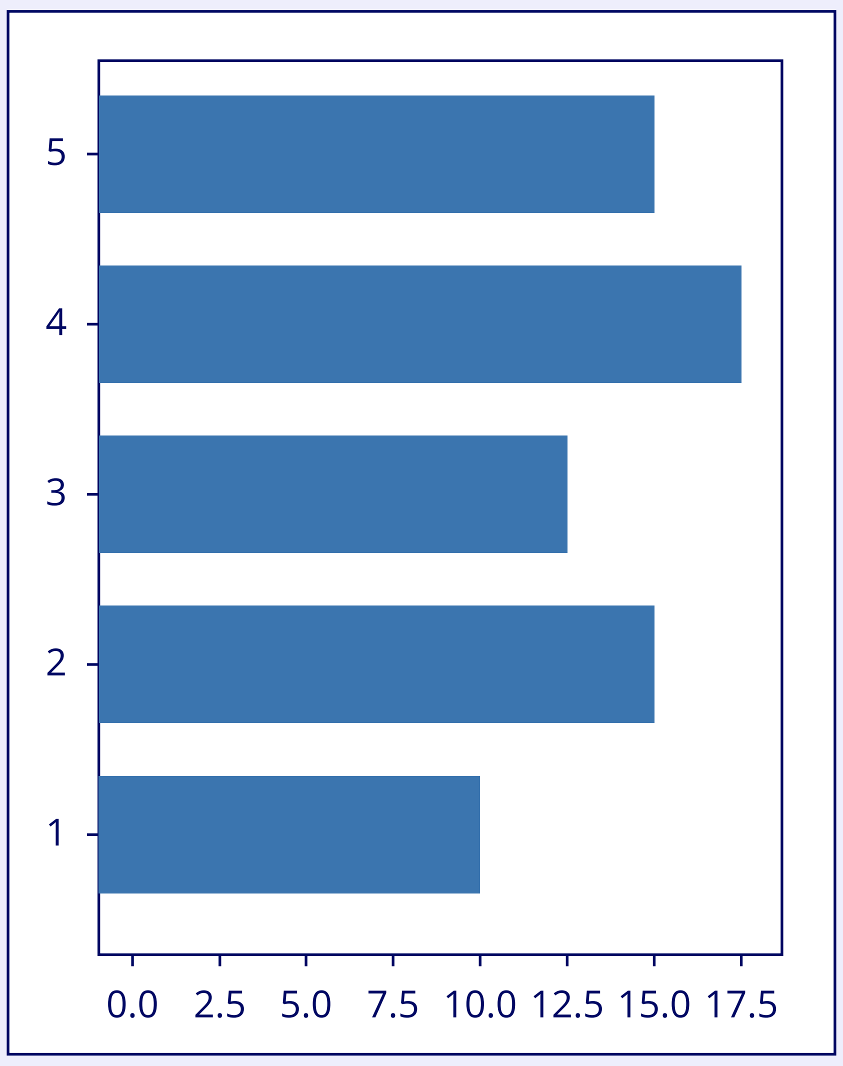
We can use a combination of plots in a single figure as well.

```
plt.bar(x, y)
plt.plot(x_range, th1, "--", color = '0.2')  # adding a certain threshold
```



- Horizontal Bar Plot
  Customizing width, color, and alignment

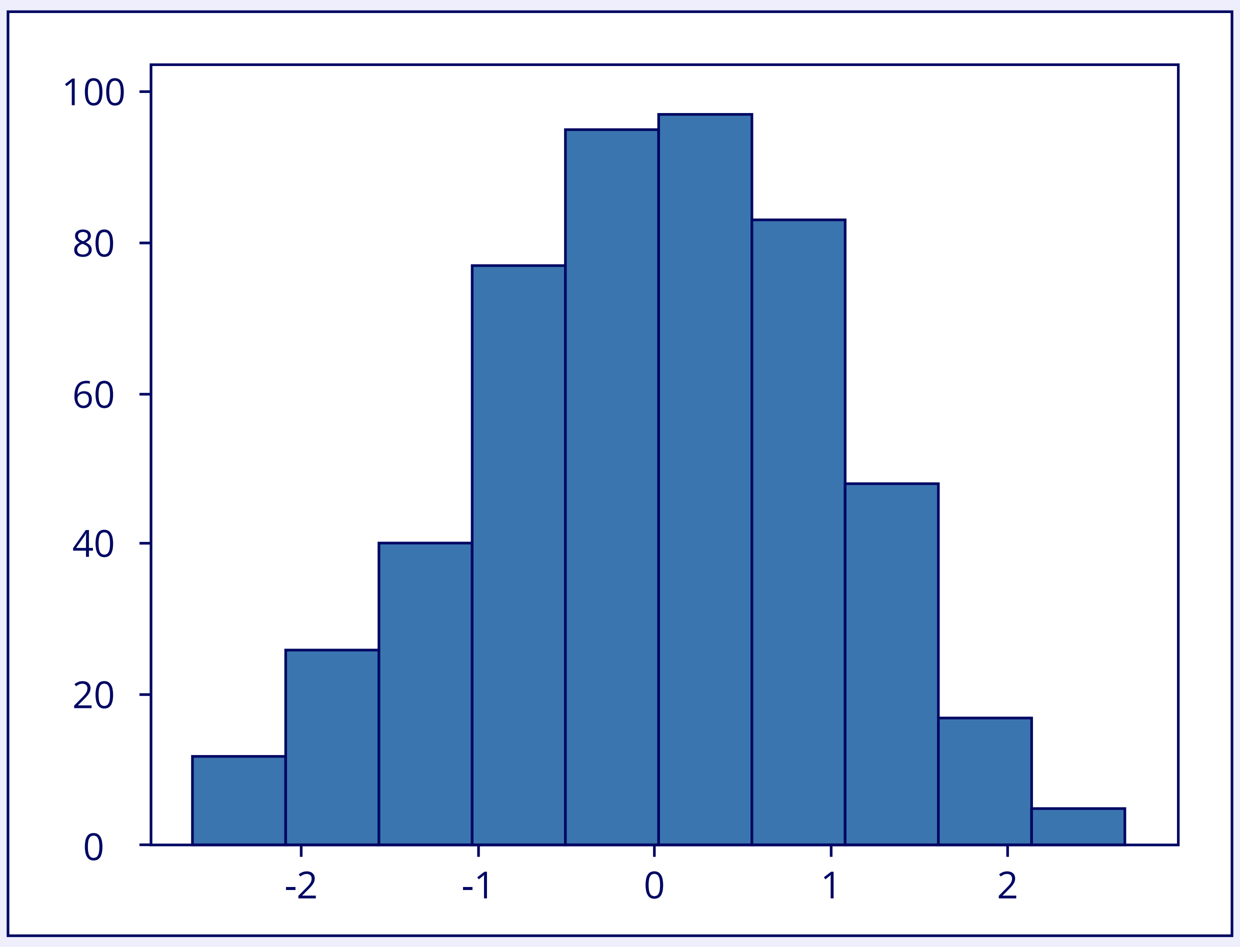
```
plt.barh(y, width)
```



Histogram

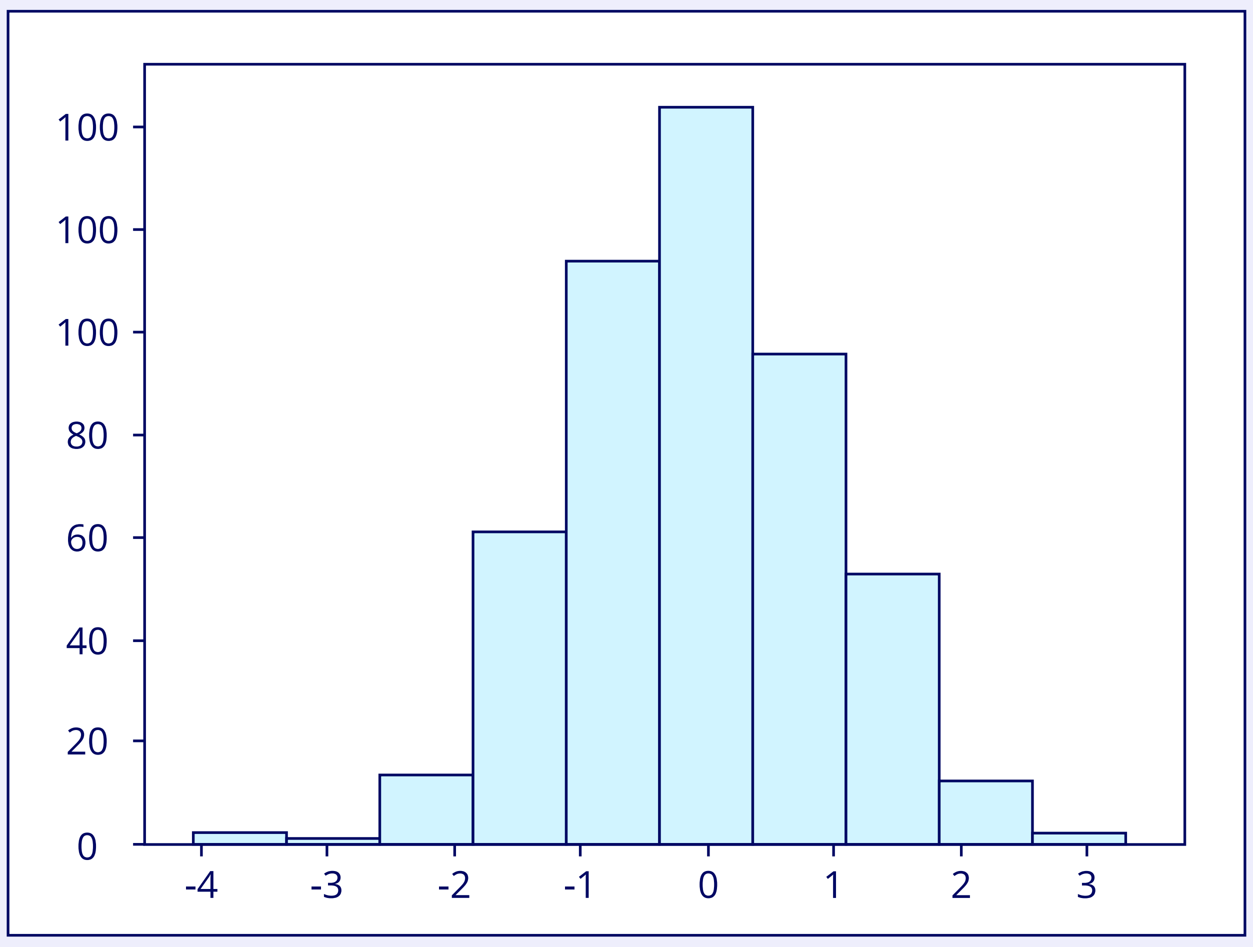
- Creating a histogram

```
plt.hist(data, bins=10)
```



- Customizing bin size and colors

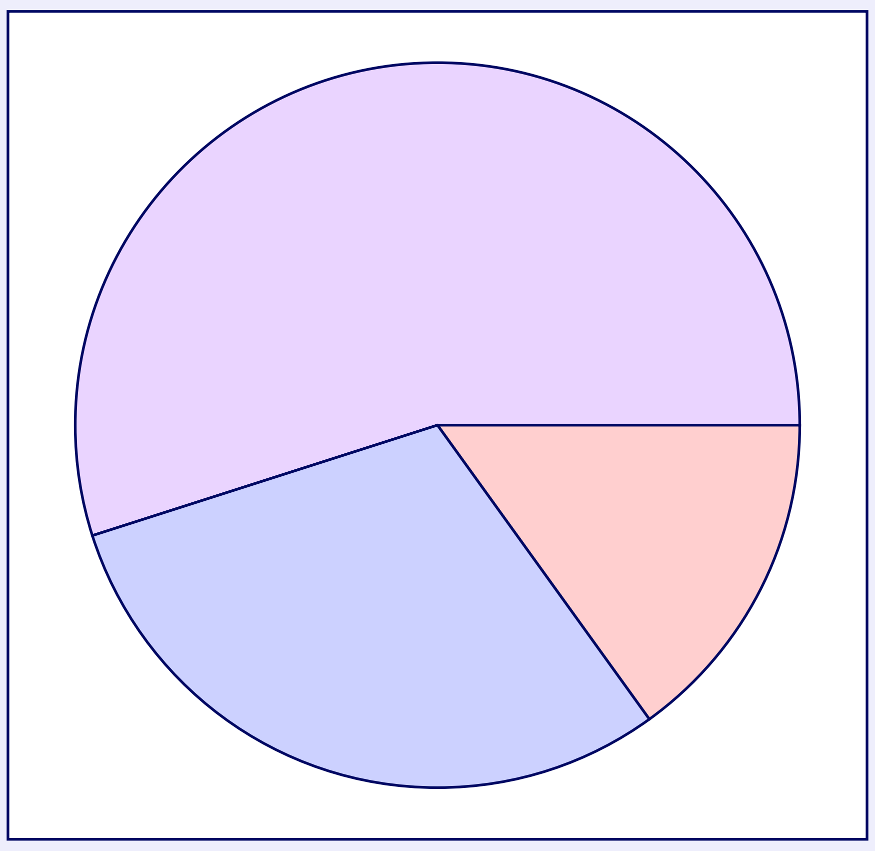
```
plt.hist(data, bins=10, color='skyblue', alpha=0.5, edgecolor='black')
```



Pie Chart

- Creating a pie chart
  sizes =[55, 30, 15]

```
plt.pie(sizes)
```

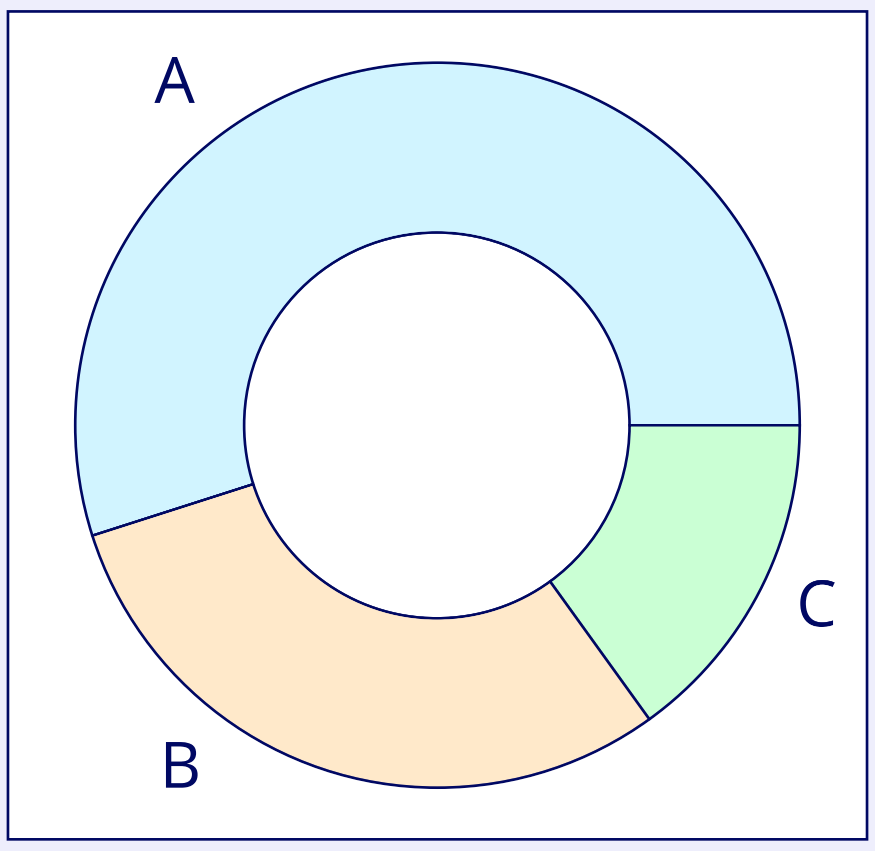


- Customizing wedge sizes and colors
  The following parameters give us a donut pie, as shown below.

```
labels = ['A', 'B', 'C']
colors=['lightcoral', 'lightskyblue', 'gold']

plt.pie(sizes, colors = colors, labels = labels)
plt.pie([1], colors = ['w'], radius = 0.5)
```

For example:



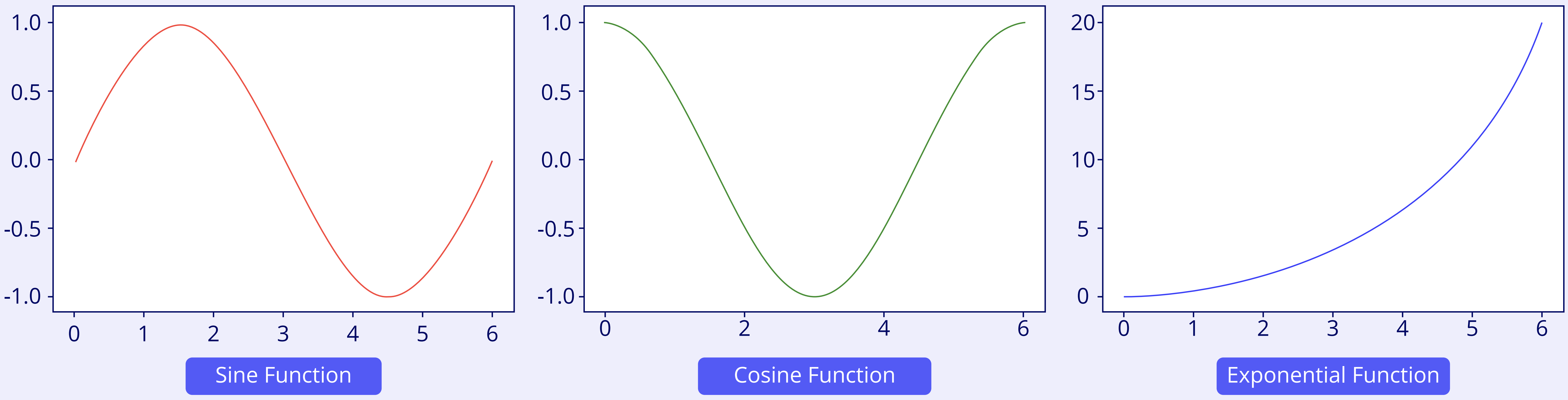
Advanced Plotting

Subplots

The **`subplots()`** function in Matplotlib creates a grid of subplots within a single figure, making it easier to compare different datasets or visualize multiple aspects of the same data.

- Creating subplots

```
# Step 1: Create some sample data
x = np.linspace(0, 2*np.pi, 400)
y1 = np.sin(x)
y2 = np.cos(x)
y3 = np.exp(x/2) # Exponential function
# Step 2: Create a figure and a grid of subplots with different sizes
fig, axs = plt.subplots(2, 2, figsize=(10, 8),
gridspec_kw={'width_ratios': [2, 1], 'height_ratios': [1, 2]})
# Step 3: Plot the first subplot in the first row
axs[0, 0].plot(x, y1, color='r')
axs[0, 0].set_title('Sine Function')
# Step 4: Plot the second subplot in the first row
axs[0, 1].plot(x, y2, color='g')
axs[0, 1].set_title('Cosine Function')
# Step 5: Plot the third subplot in the second row
axs[1, 0].plot(x, y3, color='b')
axs[1, 0].set_title('Exponential Function')
# Step 6: Remove the empty subplot in the second row and the second column
fig.delaxes(axs[1, 1])
# Step 7: Adjust the spacing between subplots
plt.tight_layout()
# Step 8: Add a main title to the entire figure
fig.suptitle('Grid of Trigonometric Functions with Different Sizes', fontsize=16, y=1.05)
```



Sine Function

Cosine Function

Exponential Function

Tips

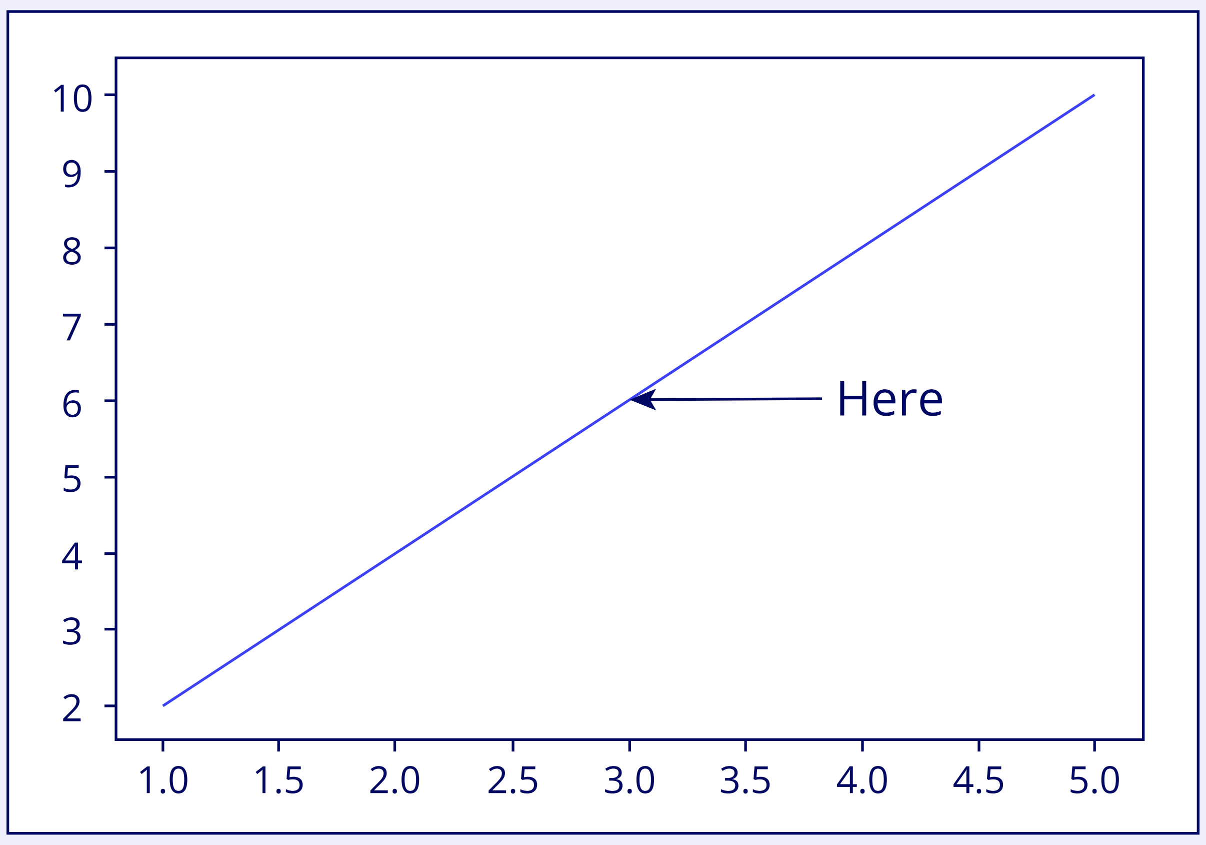
- **Adjusting subplot size:** Use **`figsize`** to control the overall size of the figure and **`gridspec_kw`** to adjust the relative sizes of the subplots.

- **Removing subplots:** Use **`fig.delaxes(ax)`** to remove unwanted subplots.
- **Spacing and titles:** The **`plt.tight_layout`** helps to automatically adjust subplot parameters to give specified padding, and **`fig.suptitle`** adds a main title to the figure.

Adding Text Annotations

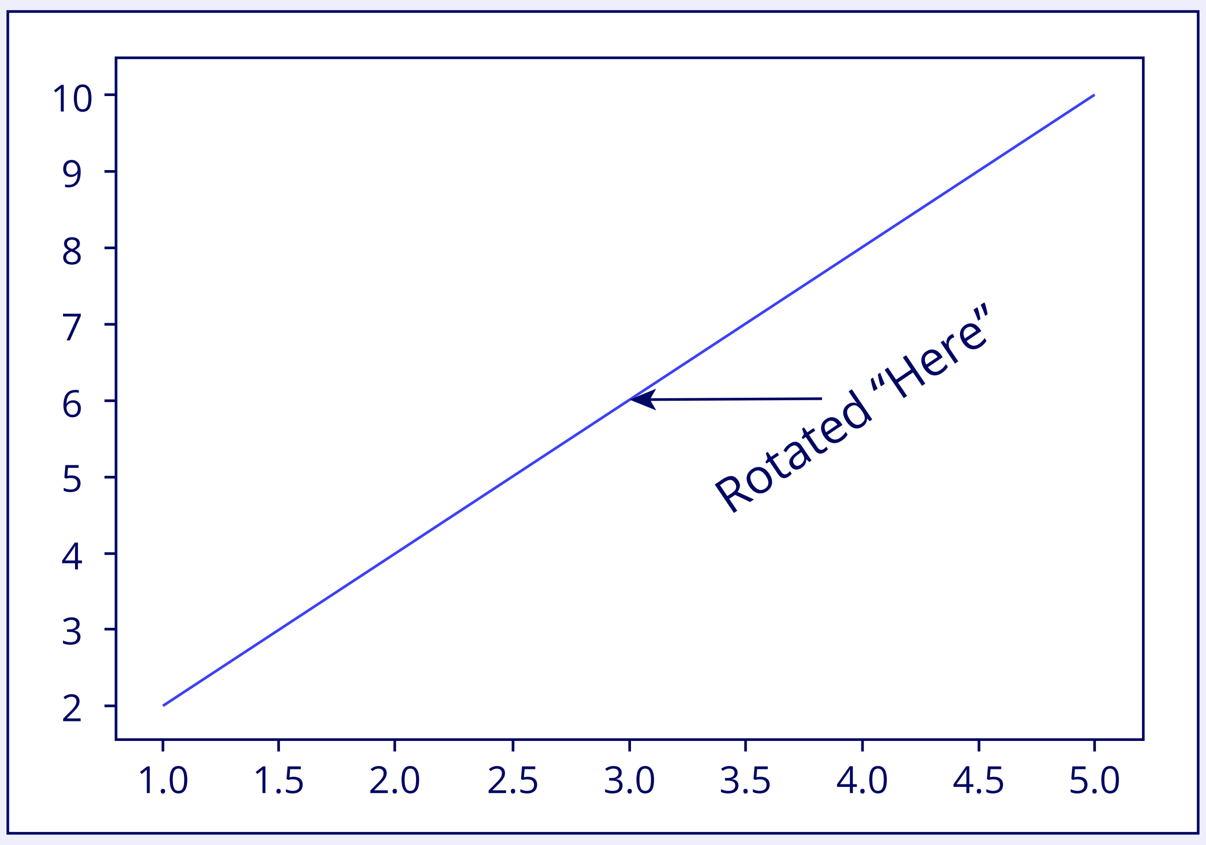
Text annotations are useful for enhancing the readability and interpretability of your plots. Use **`plt.annotate()`** to add annotations to your plot to label specific data points, highlight important features, or explain certain aspects of the plot, such as labeling maximum points.

```
plt.annotate('Here', xy=(3, 6), xytext=(3.5, 7),
arrowprops=dict(facecolor='black', arrowstyle='->'))
```



Rotating text:

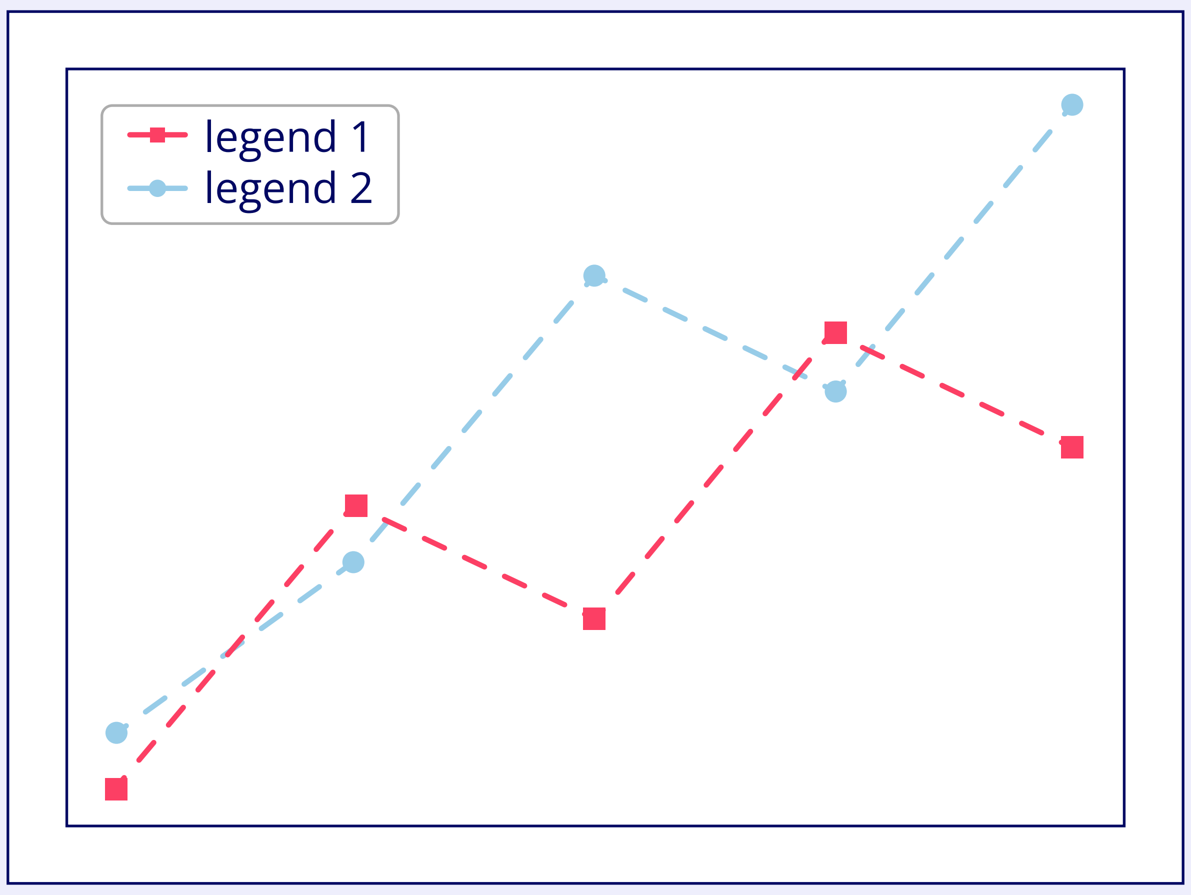
```
plt.annotate('Here', xy=(3, 6), xytext=(3.5, 7),
arrowprops=dict(facecolor='black', arrowstyle='->'), rotation = 45)
```



Adding Legends to Plots

- Adding a legend to a plot

```
plt.plot(x1, y1, label='legend 1')
plt.plot(x2, y2, label='legend  2')
plt.legend()
```



- Customizing legend location and appearance

```
plt.legend(loc='upper right')
```

Using Color Maps for Plots

- Choosing the right color map for your data

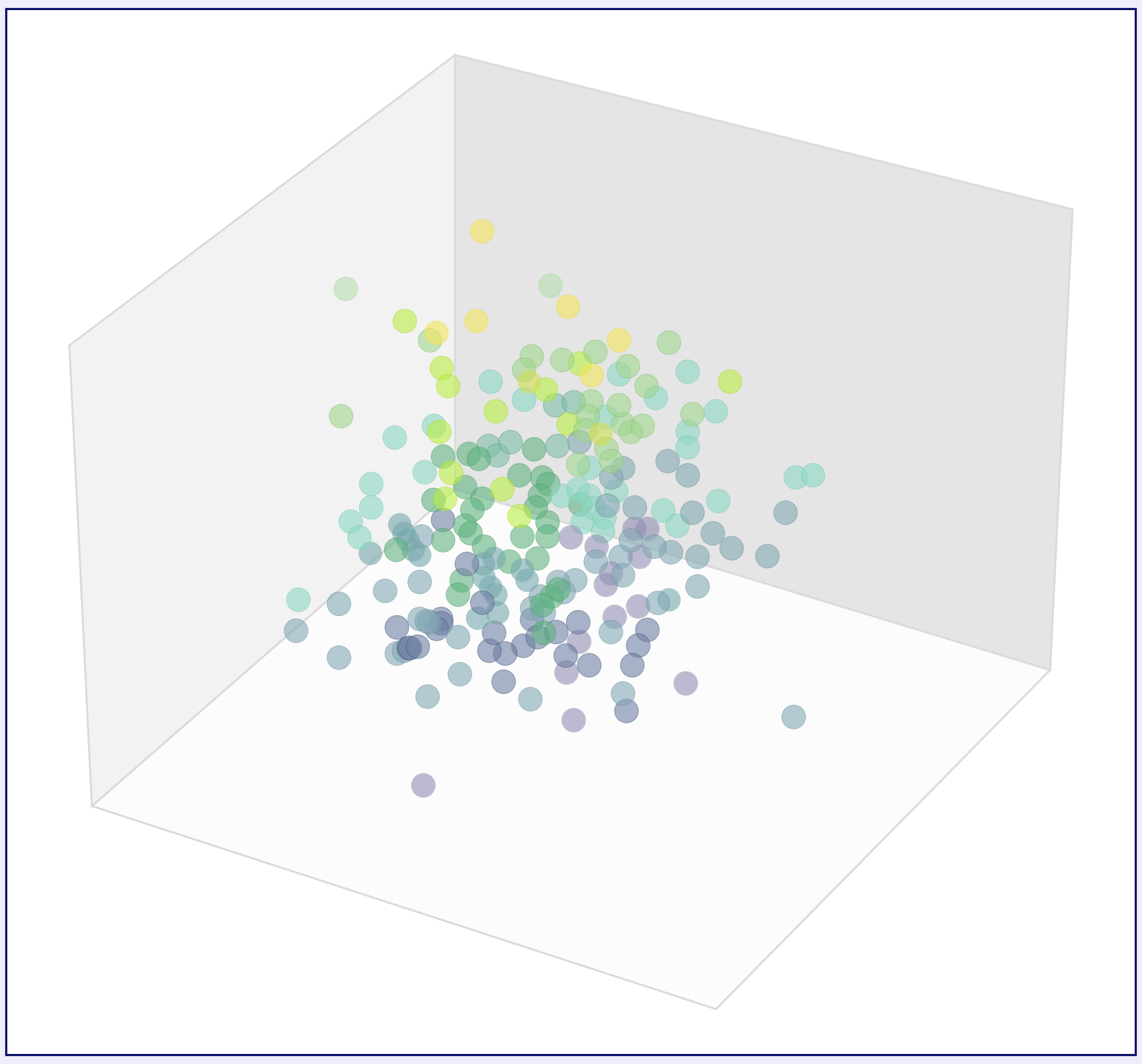
```
plt.scatter(x, y, c=z, cmap='viridis')
plt.colorbar()
```

3D Plotting

```
# Generate random data
np.random.seed(42)
x = np.random.normal(size=300)
y = np.random.normal(size=300)
z = np.random.normal(size=300)

# Create a 3D scatter plot
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

# Scatter plot
scatter = ax.scatter(x, y, z, c=z, cmap='viridis')
```



Saving Plots

- Saving plots as image files (PNG, JPEG, etc.)

```
plt.savefig('plot.png')
```

- Saving plots as PDF files

```
plt.savefig('plot.pdf')
```

Best Practices

- **Choosing the right plot type for your data**
  - Consider the nature of your data and the message you want to convey.
- **Using descriptive labels and titles**
  - Clearly label your axes and provide a descriptive title for your plot.
- **Avoiding cluttered and confusing plots**
  - Use appropriate plot types and avoid overcrowding your plots with data points.
- **Using color effectively in plots**
  - Use color to highlight important information and make your plots more visually appealing.

Performance Tips

When working with Matplotlib, follow these tips for better performance:

- **Use `plt.show()` sparingly:** Only call `plt.show()` when you need to display the plot. If creating multiple plots, consider using `plt.savefig()` to save them to a file instead of displaying them.
- **`Use the object-oriented interface:`** For more control and efficiency, use the object-oriented interface (`fig, ax = plt.subplots()`) instead of the state machine interface (`plt.plot()`).
- **Avoid unnecessary calculations:** Avoid redundant calculations by pre-calculating data outside plotting functions.
- **Use NumPy arrays:** Use NumPy arrays instead of lists for data storage and manipulation for faster operations.
- **Limit the number of data points:** For large datasets, use downsampling or aggregation to limit the number of data points plotted.
- **Use vectorized operations:** Use vectorized operations wherever possible, as they are generally faster than looping over individual data points.
- **Avoid unnecessary plot elements:** Remove plot elements such as grid lines, legends, and annotations if they are not essential for understanding the plot.
- **Use `tight_layout()`:** Use `plt.tight_layout()` to automatically adjust subplot parameters to fit the plot figure nicely.
- **Profile your code:** Use tools like `cProfile` to identify and optimize performance bottlenecks.