Essential Guide to Matplotlib

Visualize data with clarity and style: master Matplotlib with hands-on examples.

A practical and detailed guide to creating plots with Matplotlib, from simple charts to advanced figures.

rubió
Metabolomics

Ibon Martínez-Arranz | imartinez@labrubiocom
Data Science Manager at Rubió Metabolomics
www.rubiometabolomics.com

Ibon Martínez-Arranz, PhD in Mathematics and Statistics, holds MSc degrees in Applied Statistical Techniques and Mathematical Modeling. He has extensive expertise in statistical modeling and advanced data analysis techniques. Since 2017, he has led the Data Science area at Rubió Metabolomics, driving predictive model development and statistical computing management for metabolomics, data handling, and R&D projects. His doctoral research in Mathematics investigated and adapted genetic algorithms to improve the classification of NAFLD subtypes.

Itziar Mincholé Canals | iminchole@labrubio.com
Data Specialist at Rubió Metabolomics
www.rubiometabolomics.com

Itziar Mincholé Canals has a degree in physical sciences (1999) from the University of Zaragoza (Spain). In 1999 she began developing her professional career as a software analyst and developer, mainly working on web development projects and database programming in different sectors. In 2009 she joined OWL Metabolomics as a bioinformatician, where she has participated in several R&D projects and has developed several software tools for metabolomics. In 2016 she completed the master's degree in applied statistics with R software. After joining the Data Science department, she also supports data analysis.

Essential Guide to Matplotlib

Ibon Martínez-Arranz

Life Feels Good

rubió Metabolomics

Contents

Essential Guide to Matplotlib

Introduction and Setup

What is Matplotlib?

Matplotlib is the most widely used plotting library in the Python ecosystem. It provides a comprehensive interface for creating static, animated, and interactive visualizations in Python. Originally developed by John D. Hunter in 2003, it was inspired by MATLAB's plotting capabilities, aiming to provide similar functionalities in Python.

With Matplotlib, you can create a wide variety of charts, from simple line plots to complex 2D or 3D figures. It integrates well with NumPy and pandas, making it ideal for scientific computing and data analysis workflows.

Installation and Requirements

To install Matplotlib, you can use pip or conda, depending on your environment:

```
# Using pip
pip install matplotlib

# Using conda
conda install matplotlib
```

Matplotlib works with Python 3.7 or later and depends on libraries such as NumPy and cycler. If you're using a Jupyter Notebook or JupyterLab, Matplotlib is often preinstalled in data science distributions like Anaconda.

Structure of the Library: pyplot, Figure, and Axes

Matplotlib has two main interfaces:

1. **Pyplot interface** (`matplotlib.pyplot`): A state-based interface that mimics MATLAB.
2. **Object-Oriented interface**: Gives more control and flexibility by working directly with `Figure` and `Axes` objects.

Figure and Axes

- A **Figure** in Matplotlib is the entire window or page on which everything is drawn. Think of it as the canvas.
- An **Axes** is a part of the figure where data is plotted. A figure can contain multiple axes.

Here's a simple diagram of the structure:

```
1. Figure (whole canvas)
   1.1. Axes (individual plots)
        1.1.1. Axis (X and Y axis)
```

The following code snippet shows both interfaces:

```
import matplotlib.pyplot as plt

# Pyplot (state-based) interface
plt.plot([1, 2, 3], [4, 5, 6])
plt.title("Basic Line Plot")
plt.show()
```

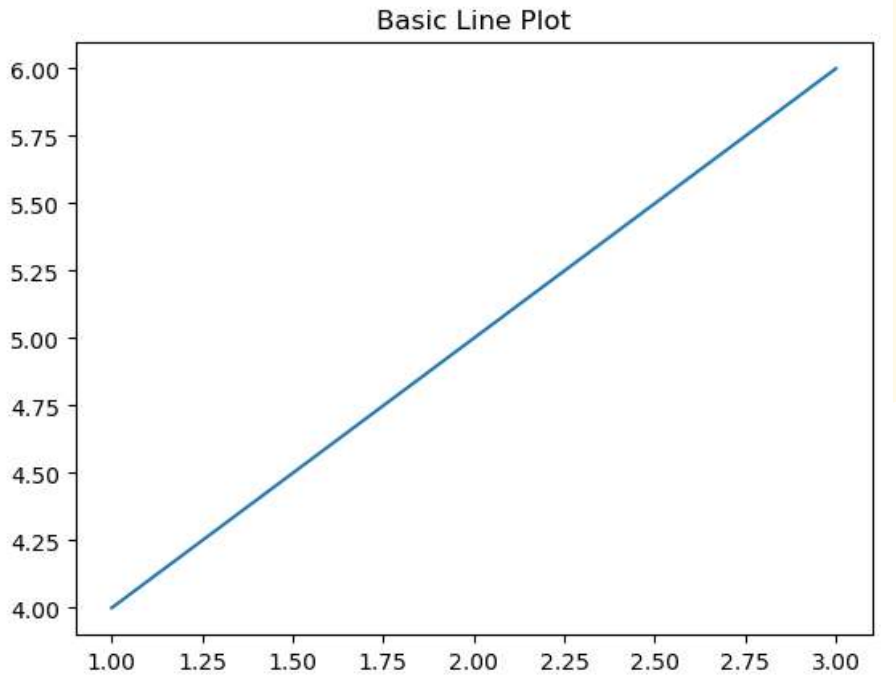

Figure 1: Image generated by the provided code.

```
# Object-Oriented interface
fig, ax = plt.subplots()
ax.plot([1, 2, 3], [4, 5, 6])
ax.set_title("Basic Line Plot (OO Interface)")
plt.show()
```

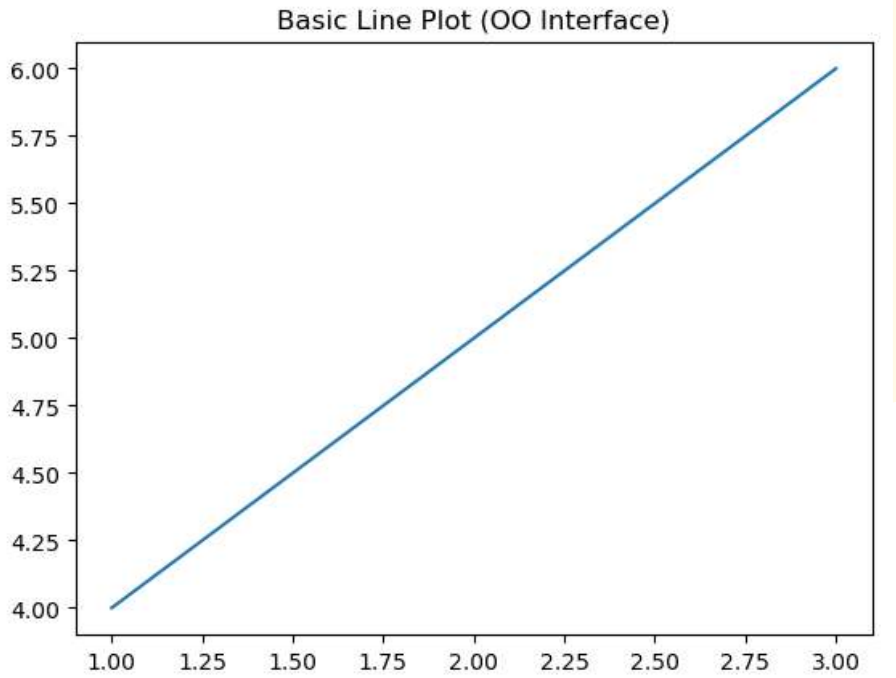

Figure 2: Image generated by the provided code.

Both approaches generate the same plot, but the object-oriented (OO) style is preferred for more complex visualizations.

First Plot: Hello, Matplotlib!

Let's create our first plot using the `pyplot` interface. We'll visualize a simple line graph.

```
1  import matplotlib.pyplot as plt
2
```

```
x = [0, 1, 2, 3, 4]
y = [0, 1, 4, 9, 16]

plt.plot(x, y)
plt.title("Hello, Matplotlib!")
plt.xlabel("x")
plt.ylabel("y")
plt.grid(True)
plt.show()
```

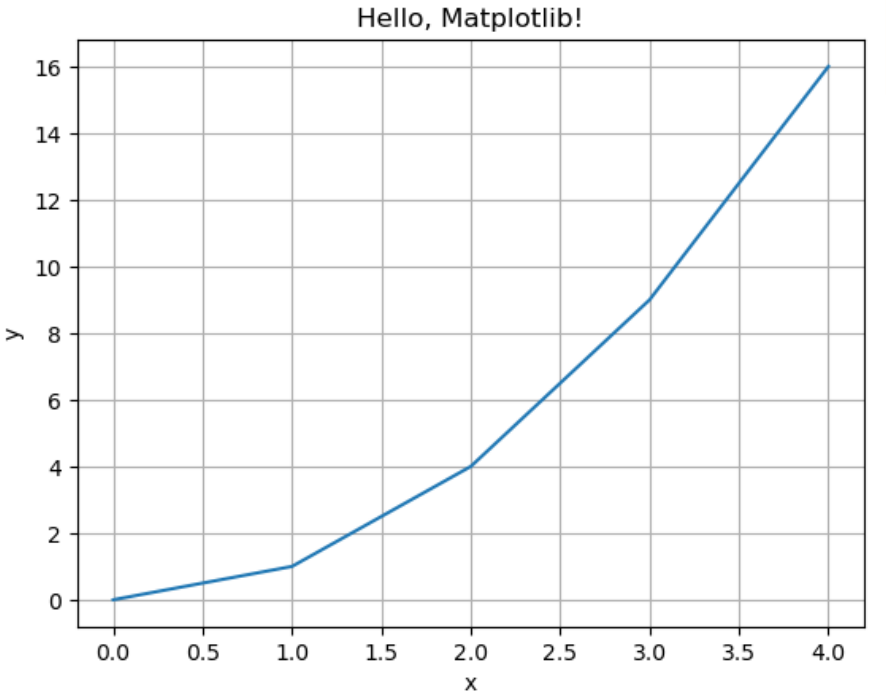

Figure 3: Image generated by the provided code.

This code will produce a basic line plot where: - `x` and `y` define the data points. - `plt.plot()` draws the line. - `plt.title()`, `plt.xlabel()` and `plt.ylabel`

`()` add labels. - `plt.grid(True)` enables a grid for better readability.

This is your starting point in the world of Matplotlib!

Basic Plotting with `pyplot`

In this chapter, we explore the most common types of plots that can be created using Matplotlib's `pyplot` interface. We'll cover **line plots**, **scatter plots**, **bar charts**, **histograms**, and **pie charts**, providing visual examples and explanations for each type.

Line Plots (`plt.plot`)

Line plots are the most basic type of chart and are commonly used to display trends over time or sequential data.

```
import matplotlib.pyplot as plt

x = [0, 1, 2, 3, 4]
y = [0, 1, 4, 9, 16]

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

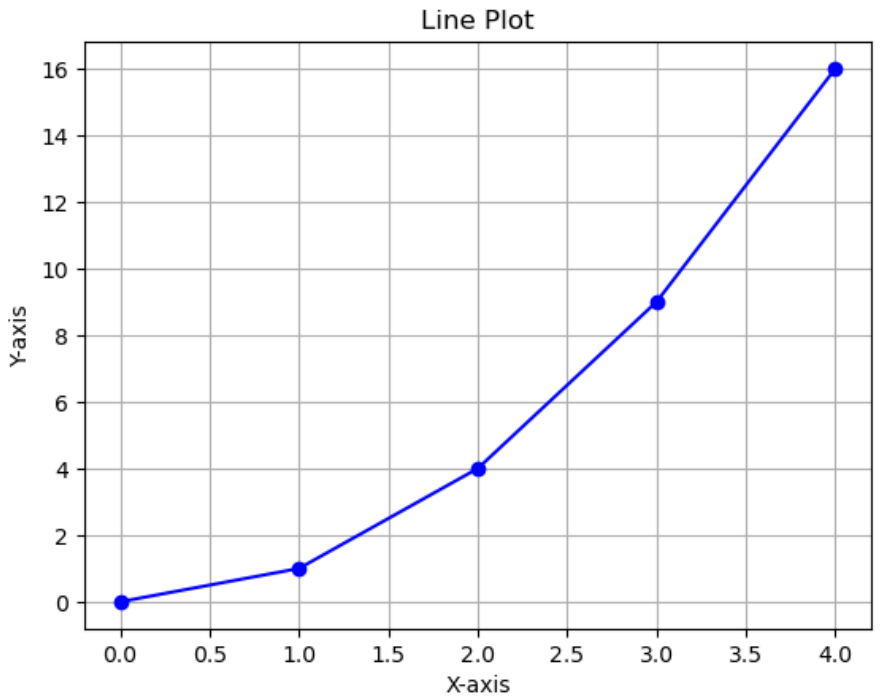

Figure 1: Image generated by the provided code.

Common arguments:

- `color`: line color (e.g., `'red'`, `'blue'`)
- `linestyle`: `'-'`, `'--'`, `':'`, `'-. '`
- `marker`: marker style at each data point (e.g., `'o'`, `'s'`, `'^'`)

Scatter Plots (`plt.scatter`)

Scatter plots are useful for visualizing the relationship between two continuous variables.

```
x = [1, 2, 3, 4, 5]
y = [5, 4, 3, 2, 1]

plt.scatter(x, y, color='green', marker='x')
plt.title("Scatter Plot")
plt.xlabel("X")
plt.ylabel("Y")
plt.grid(True)
plt.show()
```

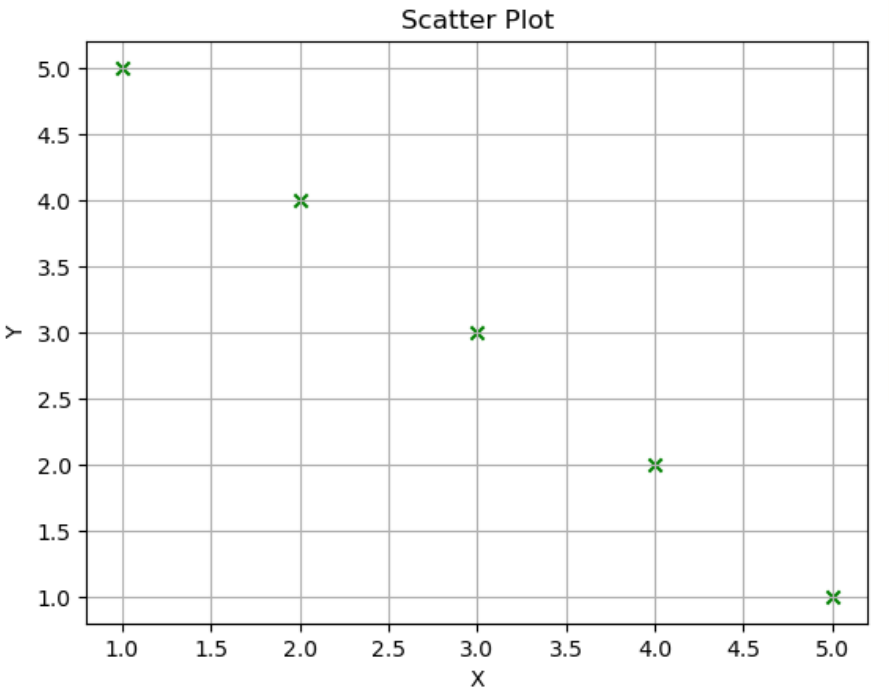

Figure 2: Image generated by the provided code.

Common arguments:

- `color`: color of the markers
- `marker`: shape of the points
- `s`: size of markers
- `alpha`: transparency (0 to 1)

Bar Charts (`plt.bar` and `plt.barh`)

Bar charts are used to show comparisons among discrete categories.

Vertical Bar Chart (`plt.bar`):

```
categories = ['A', 'B', 'C']
values = [5, 7, 3]

plt.bar(categories, values, color='purple')
plt.title("Bar Chart")
plt.ylabel("Values")
plt.show()
```

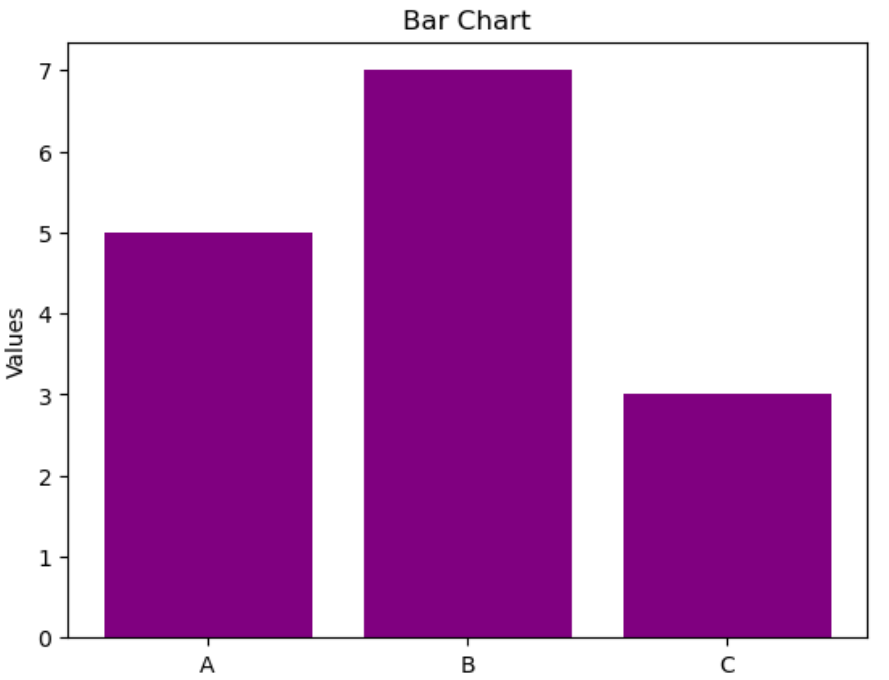

Figure 3: Image generated by the provided code.

Horizontal Bar Chart (`plt.barh`):

```
plt.barh(categories, values, color='orange')
plt.title("Horizontal Bar Chart")
plt.xlabel("Values")
plt.show()
```

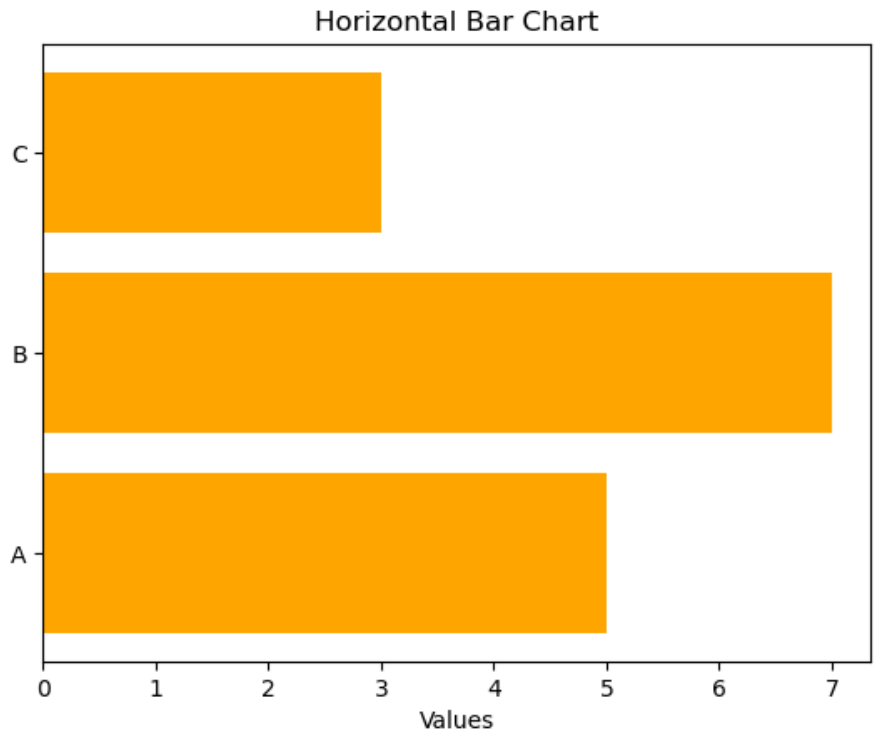

Figure 4: Image generated by the provided code.

Common arguments:

- `color`: bar color
- `width` / `height`: bar thickness
- `align`: `'center'` or `'edge'`

Histograms (`plt.hist`)

Histograms show the distribution of a dataset by grouping data into bins.

```
import numpy as np

data = np.random.randn(1000)

plt.hist(data, bins=30, color='skyblue', edgecolor='black
    ')
plt.title("Histogram")
plt.xlabel("Value")
plt.ylabel("Frequency")
plt.grid(True)
plt.show()
```

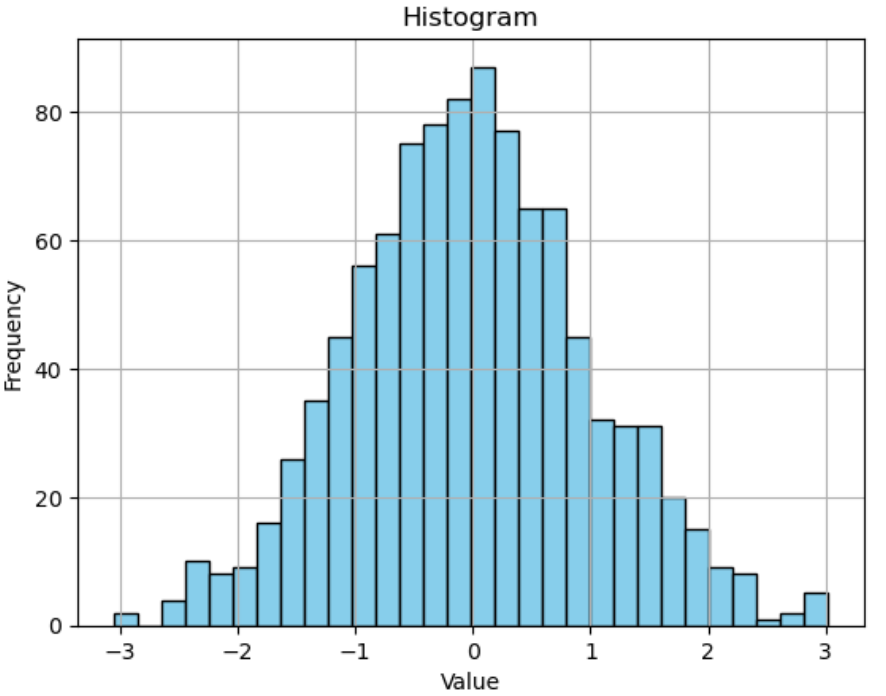

Figure 5: Image generated by the provided code.

Common arguments:

- `bins`: number of bins or bin edges
- `density`: if `True`, show probability density
- `alpha`: transparency

Pie Charts (`plt.pie`)

Pie charts show proportions of a whole. They are best used for small numbers of categories.

```
sizes = [30, 45, 25]
labels = ['Apples', 'Bananas', 'Cherries']

plt.pie(sizes, labels=labels, autopct='%1.1f%%',
    startangle=90)
plt.title("Pie Chart")
plt.axis('equal')  # Equal aspect ratio ensures pie is
    drawn as a circle.
plt.show()
```

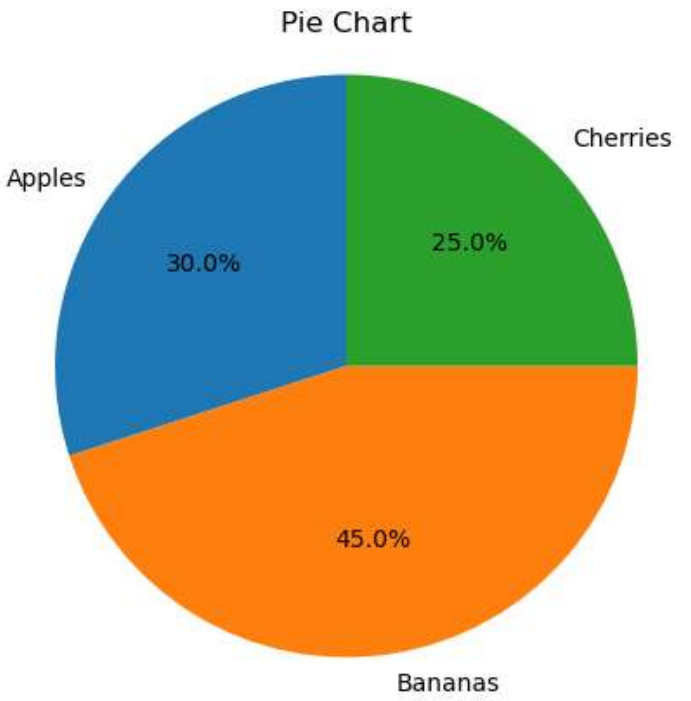

Figure 6: Image generated by the provided code.

Common arguments:

- `labels`: category labels
- `autopct`: format of value labels
- `startangle`: rotation of start angle
- `explode`: highlight slices

These fundamental plots form the basis of most data visualizations.

In the next chapters, we will explore more complex plots and learn how to fully customize them.

Understanding Figures and Axes

Matplotlib's power lies in its flexible structure built around two core objects: the **Figure** and the **Axes**. Understanding these concepts is crucial to mastering the object-oriented interface and creating complex, well-organized visualizations.

What is a Figure? What is an Axes?

- A **Figure** is the top-level container for all plot elements. It represents the entire drawing or canvas.
- An **Axes** is a part of the Figure that contains a single plot. A Figure can contain one or multiple Axes.

```
import matplotlib.pyplot as plt

fig = plt.figure()        # Create a blank Figure
ax = fig.add_subplot(1, 1, 1)  # Add one Axes to the
    Figure
ax.plot([0, 1], [0, 1])
plt.show()
```

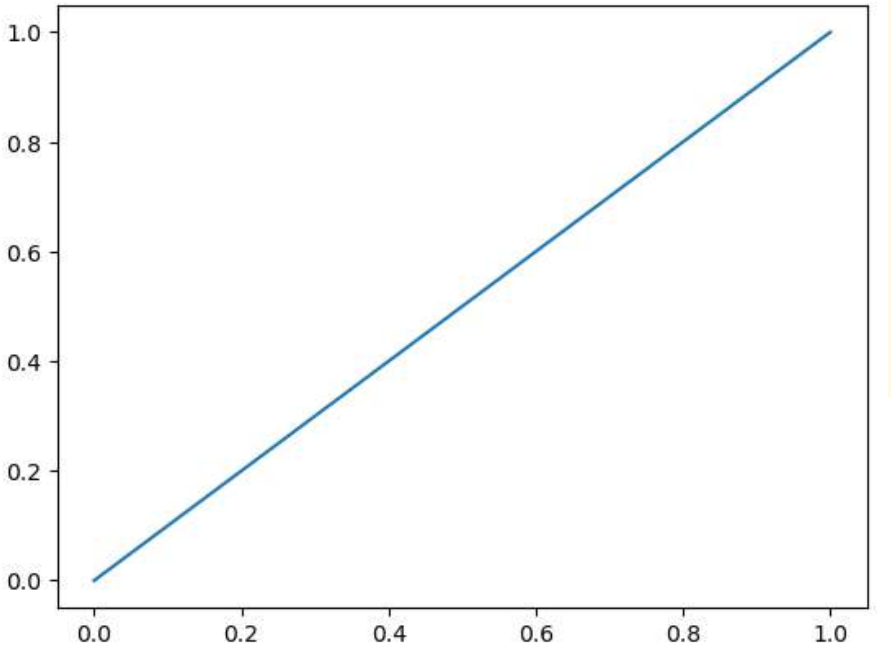

Figure 1: Image generated by the provided code.

In the above example:

- `plt.figure()` creates an empty canvas.
- `add_subplot(1, 1, 1)` means 1 row, 1 column, and selecting the 1st subplot.
- `ax.plot()` draws the line on the Axes.

Creating Multiple Subplots

You can create multiple Axes in a single Figure using either `plt.subplot()` or the more modern `plt.subplots()`.

```
fig, axs = plt.subplots(2, 2)  # 2x2 grid of subplots

axs[0, 0].plot([1, 2, 3], [1, 4, 9])
axs[0, 0].set_title("Top Left")

axs[1, 1].bar([1, 2, 3], [3, 2, 1])
axs[1, 1].set_title("Bottom Right")

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

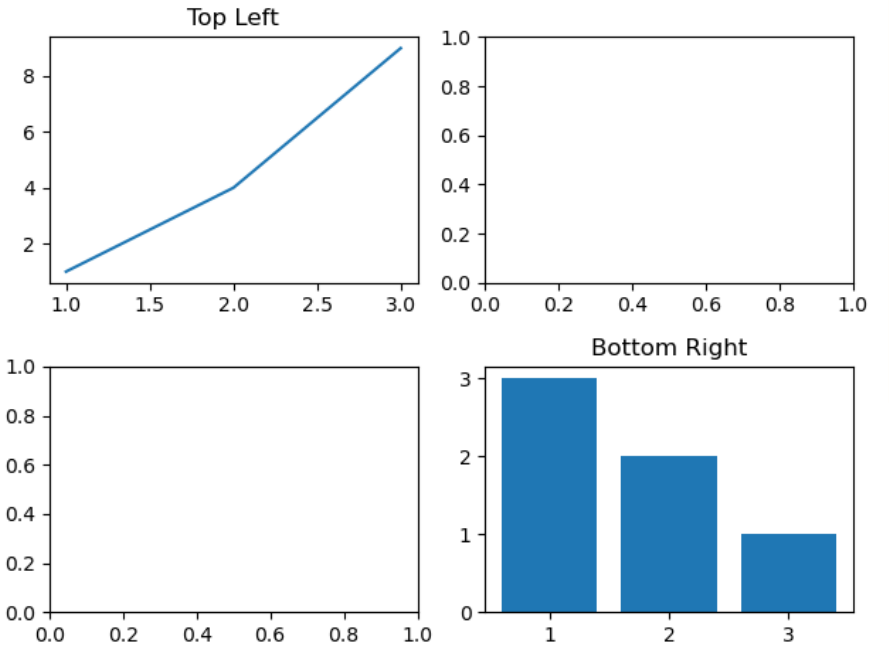

Figure 2: Image generated by the provided code.

- `axs` is a 2D NumPy array of Axes objects.
- `tight_layout()` adjusts spacing to prevent overlap.

You can also flatten the array:

```
axs = axs.flatten()
```

To loop over all subplots.

The Object-Oriented API

The object-oriented interface is more powerful and recommended for: - Creating multiple plots - Customizing individual elements - Better code structure and readability

Basic structure:

```
fig, ax = plt.subplots()
ax.plot([1, 2, 3], [1, 4, 9])
ax.set_title("Object-Oriented Plot")
ax.set_xlabel("X")
ax.set_ylabel("Y")
plt.show()
```

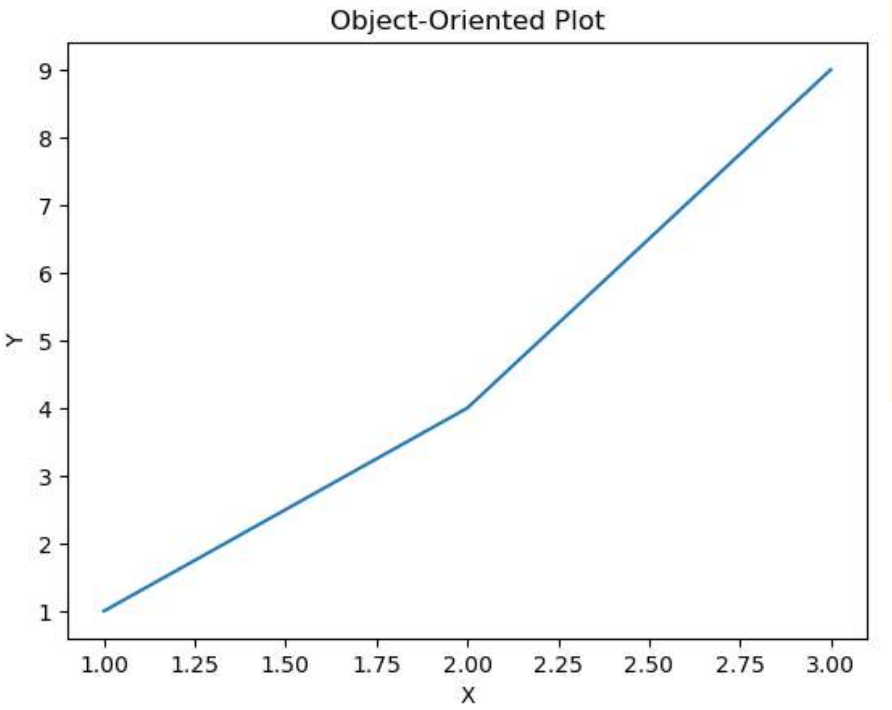

Figure 3: Image generated by the provided code.

You can access all elements (lines, ticks, labels, etc.) via the `ax` object.

`plt.subplots()` vs `plt.subplot()`

Feature	`plt.subplot()`	`plt.subplots()`
Simplicity	Good for quick, single Axes	Best for creating multiple Axes

Feature	`plt.subplot()`	`plt.subplots()`
Return value	Returns a single Axes	Returns (Figure, Axes) tuple
Recommended	Legacy / not very flexible	Preferred in most modern code

Example using `plt.subplot()`:

```
plt.subplot(1, 2, 1)
plt.plot([1, 2, 3], [3, 2, 1])
plt.title("Left Plot")

plt.subplot(1, 2, 2)
plt.plot([1, 2, 3], [1, 2, 3])
plt.title("Right Plot")

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

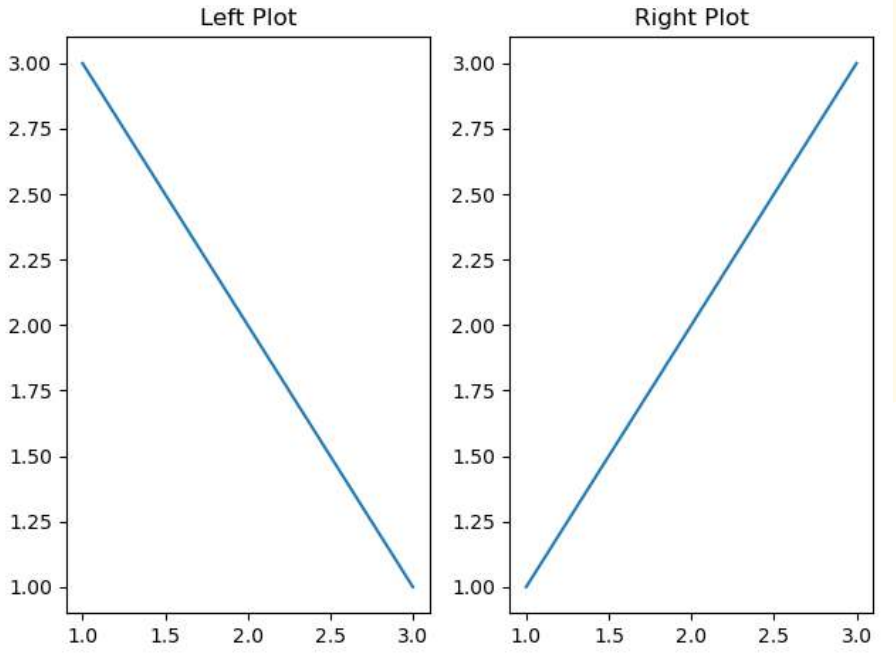

Figure 4: Image generated by the provided code.

While `plt.subplot()` is useful for quick plots, `plt.subplots()` provides better layout control and access to the full figure object.

Mastering Figures and Axes allows you to build highly customized and multi-panel visualizations.

> In the next chapters, we'll use this structure extensively as we explore plot customization and styling.

Customizing Plots: Labels, Titles, and Legends

Creating informative and readable plots often requires more than just displaying data. Adding **titles**, **axis labels**, **legends**, **tick customization**, and other elements improves both clarity and impact. This chapter shows how to customize these elements using Matplotlib's `pyplot` and object-oriented APIs.

Titles, Axis Labels, and Text Annotations

Plot Title

```
plt.title("Sample Plot")
```

Or, using the object-oriented API:

```
ax.set_title("Sample Plot")
```

Axis Labels

```
plt.xlabel("X-axis label")
plt.ylabel("Y-axis label")
```

Or with objects:

```
ax.set_xlabel("X-axis label")
ax.set_ylabel("Y-axis label")
```

Adding Text Annotations

You can add text anywhere on the plot:

```
plt.text(2, 15, "This is a point", fontsize=12, color='
    red')
```

Useful arguments:

- `fontsize`, `color`, `ha`, `va` (horizontal/vertical alignment)

Legends and Automatic Labeling

Using Labels in `plot()`

```
plt.plot(x, y1, label="Linear")
plt.plot(x, y2, label="Quadratic")
plt.legend()
```

Legend Location

You can position the legend with:

```
plt.legend(loc="upper left")
```

Some valid locations: `'best'`, `'upper right'`, `'lower left'`, `'center'`.

Customizing Legend

```
plt.legend(title="Legend Title", frameon=True, fontsize='small')
```

Tick Marks and Tick Labels

Custom Tick Locations

```
plt.xticks([0, 1, 2, 3])
plt.yticks([0, 10, 20])
```

Custom Tick Labels

```
plt.xticks([0, 1, 2], ['A', 'B', 'C'])
```

With the object interface:

```
ax.set_xticks([0, 1, 2])
ax.set_xticklabels(['A', 'B', 'C'])
```

You can also rotate tick labels:

```
plt.xticks(rotation=45)
```

Gridlines and Spines

Adding Gridlines

```
plt.grid(True)
```

Customize with:

```
plt.grid(color='gray', linestyle='--', linewidth=0.5)
```

Spines (Borders Around the Plot)

Spines are the lines that connect the axis tick marks and form the border of the data area.

```
ax.spines['top'].set_visible(False)
ax.spines['right'].set_visible(False)
```

You can also change spine colors and thickness:

```
ax.spines['left'].set_color('blue')
ax.spines['bottom'].set_linewidth(2)
```

Example: Full Customization in One Plot

The following example brings together all the elements described above to produce a clean, informative, and well-customized visualization:

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

x = npy.linspace(0, 10, 100)
y1 = x
y2 = x**2

fig, ax = plt.subplots()

# Plot lines with labels
ax.plot(x, y1, label="Linear", color='blue', linestyle='
    --', marker='o')
```

```
ax.plot(x, y2, label="Quadratic", color='green', linestyle='-', marker='s')

# Titles and labels
ax.set_title("Customized Plot Example", fontsize=14)
ax.set_xlabel("X-axis Label")
ax.set_ylabel("Y-axis Label")

# Add legend
ax.legend(title="Functions", loc="upper left", fontsize='small')

# Ticks
ax.set_xticks([0, 2, 4, 6, 8, 10])
ax.set_xticklabels(['Zero', 'Two', 'Four', 'Six', 'Eight', 'Ten'], rotation=45)
ax.set_yticks([0, 20, 40, 60, 80, 100])

# Grid and annotation
ax.grid(True, linestyle='--', alpha=0.5)
ax.text(5, 60, "Note this point", fontsize=10, color='red')

# Spine customization
ax.spines['right'].set_visible(False)
ax.spines['top'].set_visible(False)
ax.spines['left'].set_color('gray')
ax.spines['bottom'].set_linewidth(2)

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

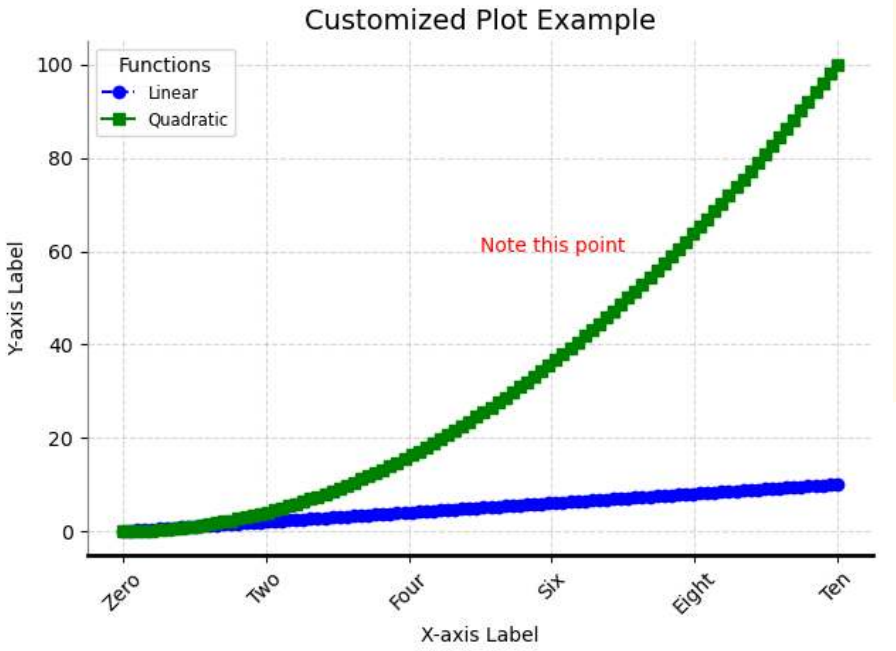

Figure 1: Image generated by the provided code.

This plot demonstrates best practices for plot customization and makes use of all the core features explained in this chapter.

These customizations give you control over the visual and semantic clarity of your plots.

> In the next chapter, we'll explore how to control the style, color, and appearance of lines and markers.

Colors, Styles, and Markers

Visual appearance is a key component of effective data visualization. In this chapter, we explore how to customize **colors**, **line styles**, **markers**, and how to use **colormaps and gradients** in Matplotlib to enhance the readability and aesthetics of your plots.

Color Specification in Matplotlib

Matplotlib supports several ways to specify colors:

Named colors

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

Short color codes

```
plt.plot(x, y, color='r')  # 'r' = red, 'g' = green, 'b' = blue, etc.
```

Hex codes

```
plt.plot(x, y, color='#FF5733')
```

RGB tuples

```
plt.plot(x, y, color=(0.1, 0.2, 0.5))  # values between 0 and 1
```

Transparency

```
plt.plot(x, y, color='blue', alpha=0.5)  # alpha controls opacity
```

Line Styles (`linestyle`, `linewidth`)

Changing the line style

```
plt.plot(x, y, linestyle='--')  # dashed line
plt.plot(x, y, linestyle='-.')  # dash-dot
plt.plot(x, y, linestyle=':')   # dotted line
```

Changing line width

```
plt.plot(x, y, linewidth=2.5)
```

Line styles can also be combined with color and markers:

```
plt.plot(x, y, color='green', linestyle='-', linewidth=2)
```

Markers: Shapes, Sizes, Colors

Markers are symbols shown at each data point. You can control their **shape**, **size**, and **color** independently from the line.

Common marker shapes:

- `'o'`: circle
- `'s'`: square
- `'^'`: triangle up
- `'D'`: diamond
- `'x'`, `'+'`: crosses

Adding markers:

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

Customizing markers:

```
plt.plot(x, y, marker='o', markersize=8, markerfacecolor=
    'white', markeredgecolor='blue')
```

Using Color Maps and Gradients

Colormaps are useful when plotting data with a third dimension, such as values represented by color.

Example with `scatter()`:

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

x = npy.random.rand(100)
y = npy.random.rand(100)
colors = npy.random.rand(100)

plt.scatter(x, y, c=colors, cmap='viridis')
plt.colorbar(label="Value")
plt.title("Colormap Example with Scatter")
plt.show()
```

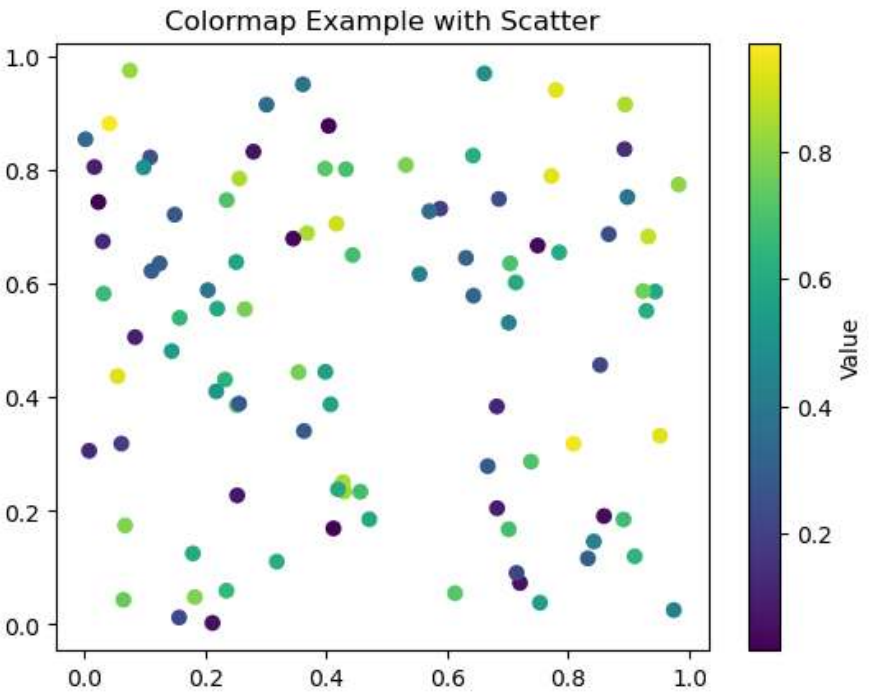

Figure 1: Image generated by the provided code.

Common colormaps:

- `'viridis'` (default)
- `'plasma'`, `'inferno'`, `'magma'`
- `'coolwarm'`, `'cividis'`, `'Blues'`, `'Greens'`

You can reverse a colormap by appending `_r`, e.g. `'viridis_r'`.

Example: Combining Colors, Styles, and Markers

The following example brings together the key elements discussed in this chapter:

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

x = np.linspace(0, 10, 100)
y1 = npy.sin(x)
y2 = npy.cos(x)
y3 = npy.exp(-0.1 * x) * np.sin(2 * x)
y4 = npy.exp(-0.1 * x) * np.cos(2 * x)

fig, axs = plt.subplots(nrows=1, ncols=2, figsize=(10, 5))

# First subplot: sin(x) and cos(x)
axs[0].plot(x, y1, label='sin(x)', color='darkorange', linestyle='--', linewidth=2,
        marker='o', markersize=6, markerfacecolor='white', markeredgecolor='darkorange')
axs[0].plot(x, y2, label='cos(x)', color='#1f77b4', linestyle='-', linewidth=2,
        marker='s', markersize=6, markerfacecolor='white', markeredgecolor='#1f77b4')
axs[0].set_title("Sine and Cosine")
axs[0].set_xlabel("x")
axs[0].set_ylabel("y")
axs[0].legend()
axs[0].grid(True, linestyle=':', linewidth=0.5, alpha=0.7)

# Second subplot: damped sine and cosine
axs[1].plot(x, y3, label='damped sin(2x)', color='mediumseagreen', linestyle='-.', linewidth=2,
        marker='^', markersize=5, markerfacecolor='yellow', markeredgecolor='black')
axs[1].plot(x, y4, label='damped cos(2x)', color='crimson', linestyle=':', linewidth=2,
```

```
            marker='d', markersize=5, markerfacecolor='
                white', markeredgecolor='crimson')
axs[1].set_title("Damped Oscillations")
axs[1].set_xlabel("x")
axs[1].set_ylabel("y")
axs[1].legend()
axs[1].grid(True, linestyle='--', linewidth=0.6, alpha
    =0.6)

plt.suptitle("Combining Colors, Styles, and Markers in
    Multiple Plots", fontsize=16)
plt.tight_layout()
plt.show()
```

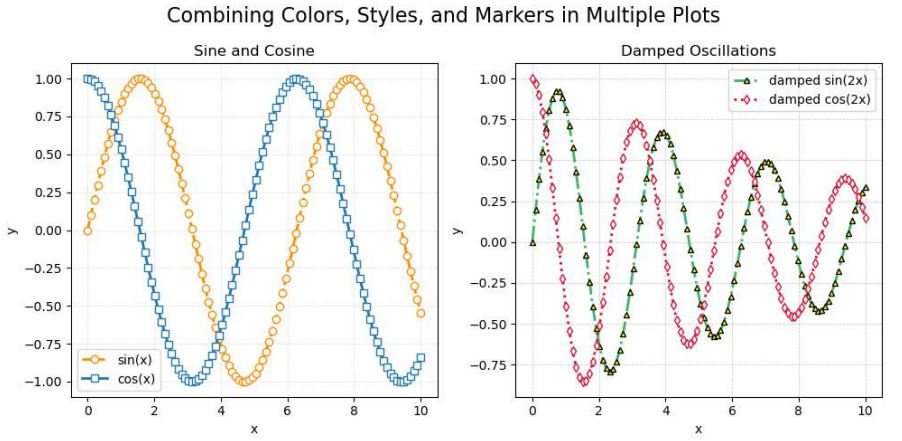

Figure 2: Image generated by the provided code.

This example uses named colors, hex colors, custom line styles, markers with styling, and a grid to illustrate how different visual elements can be combined for a professional and informative plot.

These styling options allow you to make your plots both attractive and effective in conveying information.

In the next chapter, we will look into more advanced plot types such as boxplots, heatmaps, and error bars.

Advanced Plot Types

Matplotlib supports a wide variety of plot types beyond the basics. These advanced plots are useful for statistical analysis, visualizing distributions, representing uncertainty, and exploring multidimensional data. In this chapter, we explore **error bars**, **boxplots**, **violin plots**, **heatmaps**, **contour plots**, **filled plots**, and **polar plots**.

Error Bars (`plt.errorbar`)

Error bars represent the uncertainty or variability of the data.

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

x = np.linspace(0, 10, 10)
y = np.sin(x)
error = 0.2 + 0.2 * np.sqrt(x)

plt.errorbar(x, y, yerr=error, fmt='o-', color='purple',
    ecolor='gray', capsize=4)
plt.title("Error Bars Example")
plt.xlabel("x")
plt.ylabel("y")
plt.grid(True)
plt.show()
```

Boxplots (`plt.boxplot`)

Boxplots summarize the distribution of data, highlighting the median, quartiles, and potential outliers.

```
data = [np.random.normal(0, std, 100) for std in range(1, 4)]

plt.boxplot(data, patch_artist=True)
plt.title("Boxplot Example")
plt.xlabel("Group")
plt.ylabel("Value")
plt.grid(True)
plt.show()
```

Use `patch_artist=True` to fill boxes with color.

Violin Plots (`plt.violinplot`)

Violin plots show the distribution of data across different categories, similar to boxplots but with a density estimation.

```
data = [np.random.normal(0, std, 100) for std in range(1, 4)]

plt.violinplot(data, showmeans=True, showmedians=True)
plt.title("Violin Plot Example")
plt.xlabel("Group")
plt.ylabel("Value")
plt.grid(True)
plt.show()
```

Heatmaps (`plt.imshow`, `plt.matshow`)

Heatmaps are useful for visualizing matrix-like data, such as correlations or intensities.

```
matrix = np.random.rand(10, 10)

plt.imshow(matrix, cmap='hot', interpolation='nearest')
plt.title("Heatmap with imshow")
plt.colorbar()
plt.show()
```

```
plt.matshow(matrix, cmap='Blues')
plt.title("Heatmap with matshow")
plt.colorbar()
plt.show()
```

Contour Plots (`plt.contour`, `plt.contourf`)

Contour plots are used to represent 3D data in 2D using level curves.

```
x = np.linspace(-3, 3, 100)
y = np.linspace(-3, 3, 100)
X, Y = np.meshgrid(x, y)
Z = np.exp(-X**2 - Y**2)

plt.contour(X, Y, Z, levels=10, colors='black')
plt.contourf(X, Y, Z, levels=10, cmap='viridis')
plt.title("Contour Plot")
plt.colorbar()
plt.show()
```

Filled Plots (`plt.fill_between`)

Filled plots highlight the area between curves, often used to show uncertainty or ranges.

```
x = np.linspace(0, 10, 100)
y = np.sin(x)
error = 0.3

plt.plot(x, y, color='blue')
plt.fill_between(x, y - error, y + error, color='blue',
    alpha=0.2)
plt.title("Filled Plot Example")
plt.xlabel("x")
plt.ylabel("y $\pm$ error")
plt.grid(True)
plt.show()
```

Polar Plots and Pie Charts

Polar Plot

```
theta = np.linspace(0, 2 * np.pi, 100)
r = np.abs(np.sin(5 * theta))

plt.polar(theta, r, color='teal')
plt.title("Polar Plot")
plt.show()
```

Pie Chart (revisited)

```
sizes = [25, 35, 40]
labels = ['A', 'B', 'C']

```

```
plt.pie(sizes, labels=labels, autopct='%1.1f%%',
    startangle=90)
plt.axis('equal')
plt.title("Pie Chart")
plt.show()
```

Example: Advanced Plot Grid

To wrap up this chapter, here's a combined figure showing six different advanced plot types in a 2x3 layout:

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

fig, axs = plt.subplots(2, 3, figsize=(15, 8))

# Subplot 1: Error bar plot
x = np.linspace(0, 10, 10)
y = np.sin(x)
error = 0.2 + 0.2 * np.sqrt(x)
axs[0, 0].errorbar(x, y, yerr=error, fmt='o-', color='
    purple', ecolor='gray', capsize=4)
axs[0, 0].set_title("Error Bars")

# Subplot 2: Boxplot
data = [np.random.normal(0, std, 100) for std in range(1,
     4)]
axs[0, 1].boxplot(data, patch_artist=True)
axs[0, 1].set_title("Boxplot")

# Subplot 3: Violin plot
axs[0, 2].violinplot(data, showmeans=True, showmedians=
    True)
axs[0, 2].set_title("Violin Plot")

# Subplot 4: Heatmap
matrix = np.random.rand(10, 10)
```

```
cax = axs[1, 0].imshow(matrix, cmap='coolwarm', interpolation='nearest')
axs[1, 0].set_title("Heatmap")
fig.colorbar(cax, ax=axs[1, 0], shrink=0.7)

# Subplot 5: Contour plot
x = np.linspace(-3, 3, 100)
y = np.linspace(-3, 3, 100)
X, Y = np.meshgrid(x, y)
Z = np.exp(-X**2 - Y**2)
contour = axs[1, 1].contourf(X, Y, Z, levels=10, cmap='viridis')
axs[1, 1].set_title("Contour Plot")
fig.colorbar(contour, ax=axs[1, 1], shrink=0.7)

# Subplot 6: Filled area plot
x = np.linspace(0, 10, 100)
y = np.sin(x)
axs[1, 2].plot(x, y, color='blue')
axs[1, 2].fill_between(x, y - 0.3, y + 0.3, color='blue', alpha=0.2)
axs[1, 2].set_title("Filled Plot")

plt.suptitle("Advanced Plot Types Overview", fontsize=16)
plt.tight_layout()
plt.show()
```

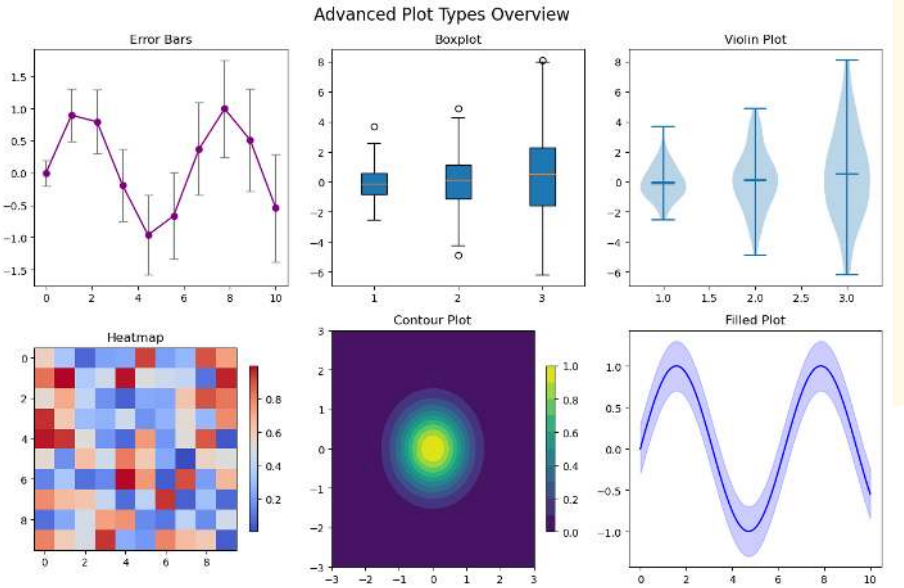

Figure 1: Image generated by the provided code.

This summary plot showcases different figure types in one go, helping you compare their usage and visual features.

These advanced plots allow you to explore data distributions, variabilities, patterns, and relationships more effectively.

> In the next chapter, we'll explore how to handle time series data in Matplotlib.

Working with Dates and Times

Handling dates and times is an essential part of many data visualization tasks, especially for time series data. Matplotlib provides tools via `matplotlib.dates` to parse, format, and customize date axes.

In this chapter, we will cover:

- Plotting time series data
- Formatting and rotating date labels
- Customizing tick frequency and appearance

Time Series with `matplotlib.dates`

We use `matplotlib.dates` (aliased as `mdates`) to work with datetime objects. First, let's create a sample time series:

```
import matplotlib.pyplot as plt
import matplotlib.dates as mdates
import numpy as np
import datetime

# Generate 30 days of data
base = datetime.datetime.today()
dates = [base - datetime.timedelta(days=x) for x in range
    (30)][::-1]
values = np.random.normal(loc=0.0, scale=1.0, size=30).
    cumsum()

```

```
fig, ax = plt.subplots(figsize=(10, 5))

ax.plot(dates, values, marker='o', linestyle='-', color='tab:blue')
ax.set_title("Time Series with Dates")
ax.set_xlabel("Date")
ax.set_ylabel("Cumulative Value")

# Format x-axis with readable dates
ax.xaxis.set_major_locator(mdates.WeekdayLocator(interval=1))
ax.xaxis.set_major_formatter(mdates.DateFormatter('%b %d'))
fig.autofmt_xdate(rotation=45)

# Minor ticks per day
ax.xaxis.set_minor_locator(mdates.DayLocator())
ax.grid(True, which='both', linestyle='--', linewidth=0.5, alpha=0.7)

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

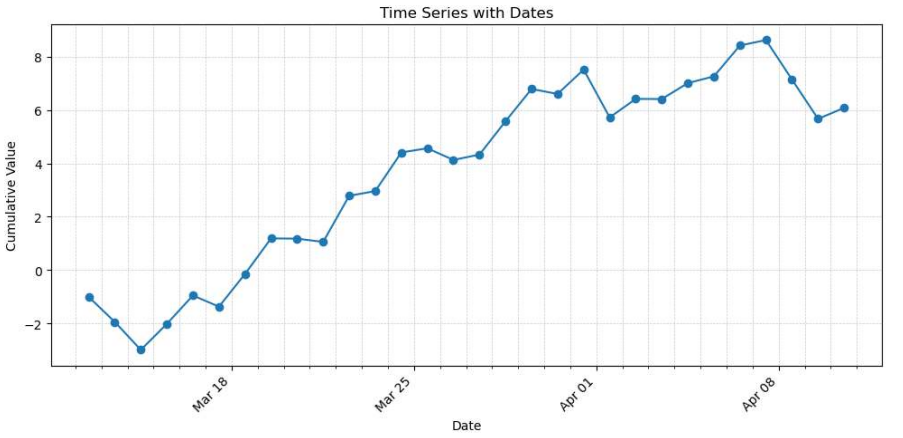

Figure 1: Image generated by the provided code.

Explanation:

- `WeekdayLocator(interval=1)`: shows one tick per week
- `DateFormatter('%b %d')`: formats dates as “Apr 10”, “Apr 17”, etc.
- `fig.autofmt_xdate(rotation=45)`: auto-rotates and aligns dates
- `DayLocator()`: adds minor ticks for each day

This example gives you full control over how dates appear and behave in your time series plots.

> In the next chapter, we’ll learn how to save and export figures with different formats and resolutions.

Saving and Exporting Figures

Once you've created a plot, you may want to save it for inclusion in reports, presentations, or publications. Matplotlib provides the `plt.savefig()` function for exporting figures in various formats with detailed control over quality, size, background, and layout.

`plt.savefig()` Formats and Options

You can save a figure using:

```
plt.savefig("filename.png")
```

Supported formats include: - PNG (default): for web and screen display - PDF: for high-quality documents - SVG: scalable vector graphics (useful for web) - JPG, EPS, TIFF (depending on backend)

You can specify the format explicitly:

```
plt.savefig("plot.pdf", format='pdf')
```

DPI and Resolution

DPI (dots per inch) controls the resolution of raster formats (like PNG or JPG). The default is usually 100.

```
plt.savefig("high_res_plot.png", dpi=300)
```

Higher DPI values produce sharper images, especially useful for print publications.

Transparent Background and Tight Layout

You can remove the background box and trim white space:

```
plt.savefig("transparent_plot.png", transparent=True,
    bbox_inches='tight')
```

Options:

- `transparent=True`: removes background fill (good for overlays)
- `bbox_inches='tight'`: reduces extra margins and padding

Embedding in PDFs and Reports

For integration into LaTeX documents or scientific reports, vector formats are preferred:

```
plt.savefig("figure.svg")
plt.savefig("figure.pdf")
```

These formats preserve quality at any zoom level.

You can also embed figures in Jupyter notebooks using the `%matplotlib inline` magic command and export the notebook as HTML, PDF or Markdown.

Example: Export a Publication-Ready Plot

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

x = npy.linspace(0, 2 * npy.pi, 100)
y = npy.sin(x)

fig, ax = plt.subplots()
ax.plot(x, y, label='sin(x)', color='crimson', linewidth=2)
ax.set_title("Sine Function")
ax.set_xlabel("x")
ax.set_ylabel("sin(x)")
ax.legend()
ax.grid(True, linestyle='--', alpha=0.6)

# Save figure with high quality
plt.savefig("sine_plot.pdf", format='pdf', dpi=300, bbox_inches='tight', transparent=True)
plt.show()
```

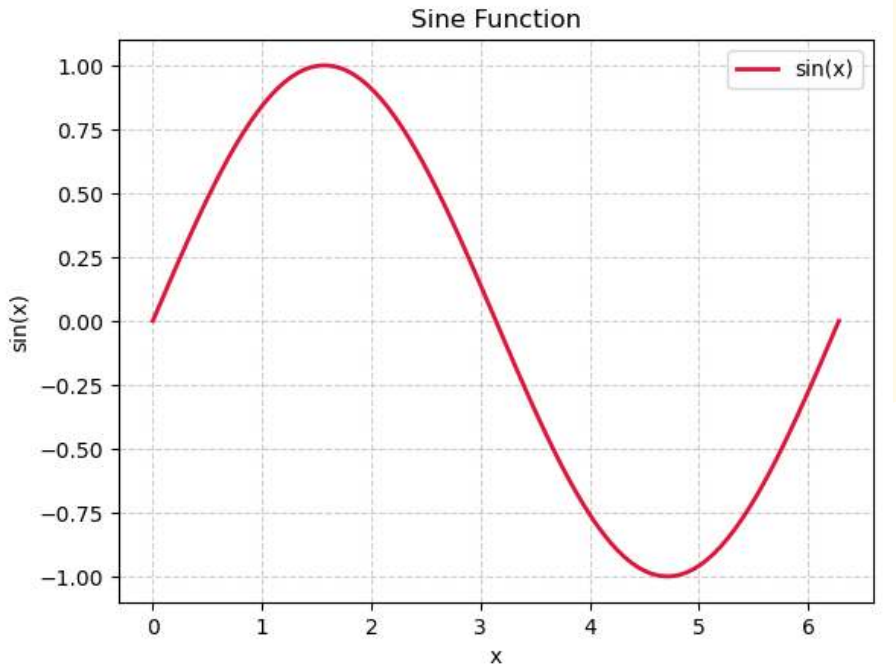

Figure 1: Image generated by the provided code.

This example creates a clean, annotated plot and saves it as a high-resolution vector PDF with no padding or background color — ideal for reports and publications.

In the next chapter, we'll explore Matplotlib's runtime configuration system, styles, and how to globally or locally define custom aesthetics.

Styling with matplotlib Configurations

Matplotlib offers powerful configuration options to control the look and feel of plots. These include global and local settings, predefined style sheets, and even the ability to define your own styles for consistent branding or publication-ready figures.

`matplotlibrc` and Runtime Configuration (`rcParams`)

The appearance of plots is controlled by the `rcParams` dictionary, which stores the current settings.

Access and modify runtime config:

```
import matplotlib.pyplot as plt

plt.rcParams['lines.linewidth'] = 3
plt.rcParams['axes.titlesize'] = 14
```

These changes will persist for the current session.

You can also use the `rc` function for more concise control:

```
plt.rc('lines', linewidth=2)
plt.rc('axes', facecolor='#f0f0f0')
```

Global vs Local Style Settings

- **Global** settings affect all plots once configured via `rcParams` or `plt.rc()`.
- **Local** settings can be defined using the `with plt.rc_context()` context manager:

```
with plt.rc_context({'lines.linewidth': 4, 'axes.
    titlesize': 18}):
    plt.plot([0, 1, 2], [0, 1, 4])
    plt.title("Local Style Example")
```

Outside the `with` block, settings return to previous values.

Predefined Styles

Matplotlib comes with a set of built-in style sheets. You can list them with:

```
print(plt.style.available)
```

Some common styles:

- `'default'`
- `'ggplot'`
- `'seaborn'`
- `'fivethirtyeight'`
- `'bmh'`

Apply a style globally:

```
plt.style.use('seaborn-v0_8')  # or 'ggplot', 'bmh', etc.
```

Apply a style temporarily:

```
with plt.style.context('ggplot'):
    plt.plot([1, 2, 3], [1, 4, 9])
```

Creating Your Own Style Sheet

You can create custom styles by saving configuration settings in a `.mplstyle` file.

Example: `my_style.mplstyle`

```
axes.titlesize: 16
axes.labelsize: 14
xtick.labelsize: 12
ytick.labelsize: 12

lines.linewidth: 2
lines.linestyle: -
lines.marker: o
lines.markersize: 6

axes.facecolor: F7F7F7
figure.facecolor: FFFFFF
grid.color: FF0000
text.color: 000000
axes.edgecolor: 000000

font.size: 12
font.family: sans-serif
font.sans-serif: Arial, DejaVu Sans

xtick.major.size: 6
```

```
ytick.major.size: 6
xtick.direction: in
ytick.direction: in
grid.linestyle: --
grid.linewidth: 0.5
```

Save the file and apply it with:

```
plt.style.use('my_style.mplstyle')
```

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

plt.style.use('my_style.mplstyle')
x = npy.linspace(0, 10, 100)
y1 = npy.sin(x)
plt.plot(x,y)
plt.title("Title")
plt.xlabel("xlabel")
plt.ylabel("ylabel")
plt.grid()
```

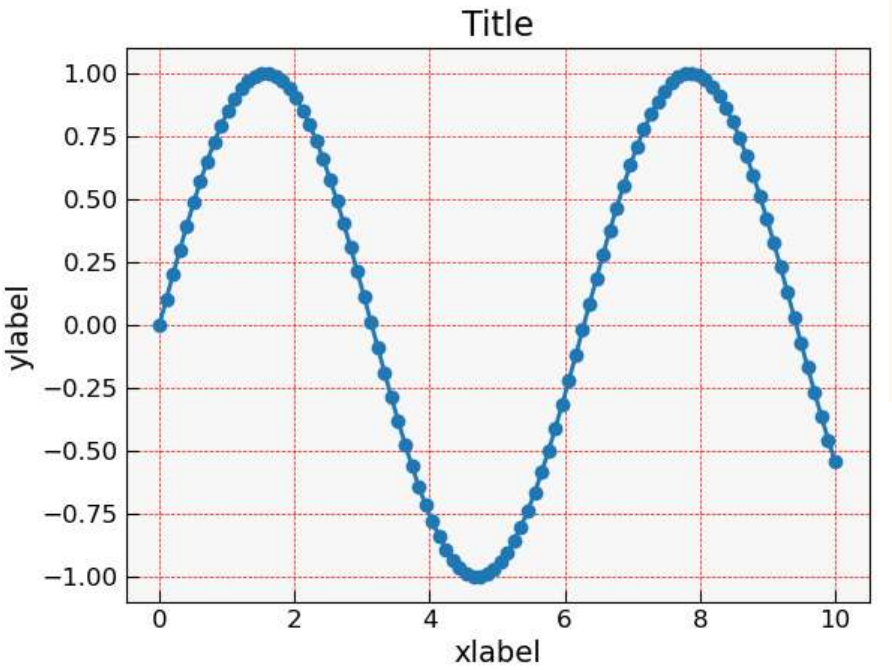

Figure 1: Image generated by the provided code.

Place it in your working directory or in `~/.config/matplotlib/stylelib/` to make it reusable.

Example: Using a Style Sheet

```
1  import matplotlib.pyplot as plt
2  import numpy as npy
3
4  x = npy.linspace(0, 10, 100)
5  y1 = npy.sin(x)
```

```
y2 = npy.cos(x)
y3 = npy.sin(x) * np.cos(x)

styles = [
    'default', 'ggplot', 'seaborn-v0_8',
    'fivethirtyeight', 'bmh', 'dark_background'
]
titles = [
    'Default', 'ggplot', 'seaborn',
    'FiveThirtyEight', 'bmh', 'Dark Background'
]

fig, axs = plt.subplots(2, 3, figsize=(15, 8))
axs = axs.flatten()

for ax, style, title in zip(axs, styles, titles):
    with plt.style.context(style):
        ax.plot(x, y1, label='sin(x)')
        ax.plot(x, y2, label='cos(x)')
        ax.plot(x, y3, label='sin(x)cos(x)')
        ax.set_title(f"Style: {title}")
        ax.set_xlabel("x")
        ax.set_ylabel("y")
        ax.legend()
        ax.grid(True)

fig.suptitle("Comparison of 6 Matplotlib Style Sheets",
    fontsize=18)
plt.tight_layout()
plt.show()
```

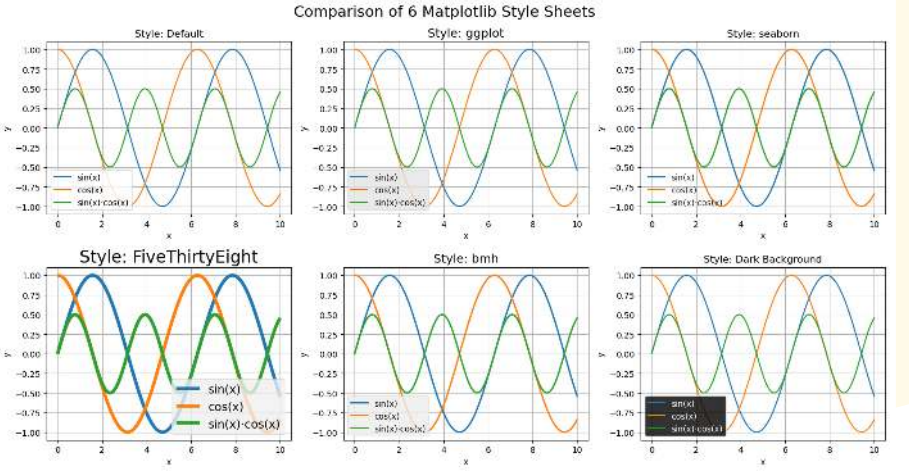

Figure 2: Image generated by the provided code.

This plot will automatically adopt the styling defined by the seaborn style sheet.

> In the next chapter, we'll look into interactive and animated plots for more dynamic visualizations.

References: Common Function Arguments

This final chapter serves as a reference guide for frequently used arguments in Matplotlib functions. It includes comparison tables across plot types, default values, and concise cheatsheets. It also provides links for further reading and a gallery of examples.

Common Plotting Arguments

Argument	Purpose	Example Value
`color`	Line/marker color	`'blue'`, `#FF5733`
`linewidth`	Thickness of the line	`2`, `0.5`
`linestyle`	Line type	`'-'`, `'--'`, `':'`
`marker`	Symbol at each data point	`'o'`, `'s'`, `'^'`
`markersize`	Size of markers	`6`, `10`
`label`	Legend entry	`'sin(x)'`
`alpha`	Transparency	`0.5`, `1.0`

Argument	Purpose	Example Value
`cmap`	Colormap for scalar values	`'viridis'`, `'plasma'`
`fmt`	Format string (for `errorbar`)	`'o--'`, `'s:'`
`bins`	Histogram bin count	`10`, `30`
`orientation`	Histogram/bar orientation	`'vertical'`, `'horizontal'`
`stacked`	Stack bars or histograms	`True`, `False`

Function-Specific Comparison Table

Plot Type	Function	Special Args
Line plot	`plt.plot`	`linestyle`, `linewidth`, `marker`
Scatter plot	`plt.scatter`	`s`, `c`, `marker`, `cmap`
Bar chart	`plt.bar`	`width`, `align`, `color`
Histogram	`plt.hist`	`bins`, `density`, `alpha`
Boxplot	`plt.boxplot`	`notch`, `patch_artist`, `vert`
Violin plot	`plt.violinplot`	`showmeans`, `showmedians`
Errorbar plot	`plt.errorbar`	`yerr`, `xerr`, `fmt`, `capsize`
Pie chart	`plt.pie`	`autopct`, `startangle`, `explode`
Heatmap	`plt.imshow`	`interpolation`, `cmap`
Contour	`plt.contourf`	`levels`, `cmap`, `alpha`

Defaults and How to Override

- Default line width: `1.5`
- Default marker size: `6`
- Default font size: `10`
- Default DPI: `100` for on-screen display

You can override globally via:

```
plt.rcParams['lines.linewidth'] = 2.5
```

Or locally via function arguments:

```
plt.plot(x, y, linewidth=2.5)
```

Appendix

Gallery of Plot Examples

You can find many categorized examples in the official Matplotlib gallery:

- https://matplotlib.org/stable/gallery/index.html

Cheatsheet and Quick Reference

Matplotlib's official cheatsheet (PDF):

- https://matplotlib.org/cheatsheets/

Further Reading and Resources

- Matplotlib Documentation

- Pyplot API Reference
- Customizing Matplotlib with rcParams
- Colormaps Reference
- Seaborn integration

This wraps up the *Essential Guide to Matplotlib*. Keep experimenting, building, and customizing — and may your plots always be sharp, informative, and beautiful!


rubió
Metabolomics