#UpSkillWithKalpesh

Day 13

Data Science Unlocked

From Zero to Data Hero

Matplotlib for Data Visualization – Part 2

Kalpesh Pathade
@DataSimplified

Matplotlib for Data Visualization - Part 2

Type	Data science masterclass

IV. Types of Plots in Matplotlib

Matplotlib provides a variety of plots to visualize different types of data. Each plot type is suited for specific use cases, from **trend analysis** (line plots) to **distribution visualization** (histograms & KDE plots).

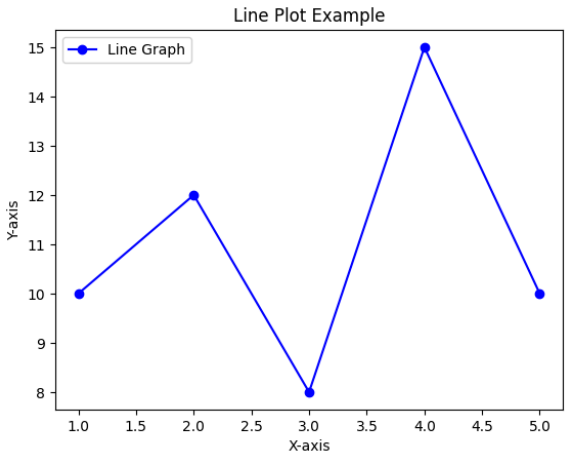
4.1 Line Graphs

Use Case:

- Best for visualizing trends over time or continuous data.

```
import matplotlib.pyplot as plt

x = [1, 2, 3, 4, 5]
y = [10, 12, 8, 15, 10]

plt.plot(x, y, marker="o", linestyle="-", color="b", label="Line Graph")
plt.xlabel("X-axis")
plt.ylabel("Y-axis")
plt.title("Line Plot Example")
plt.legend()
plt.show()
```

Line Plot Output:



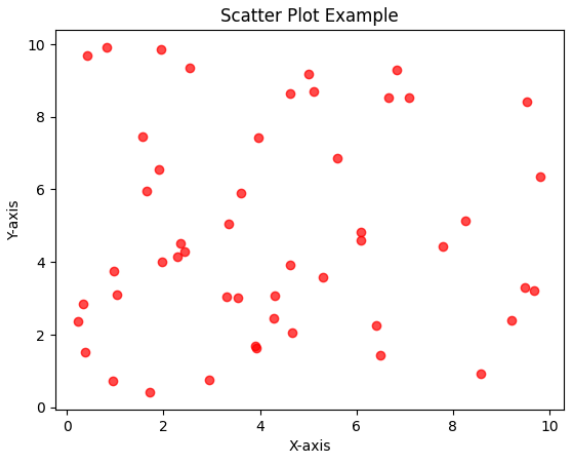
4.2 Scatter Plots

Use Case:

- Used to show relationships between two numerical variables.

```
import numpy as np

x = np.random.rand(50) * 10
y = np.random.rand(50) * 10

plt.scatter(x, y, color="red", marker="o", alpha=0.7)
plt.xlabel("X-axis")
plt.ylabel("Y-axis")
plt.title("Scatter Plot Example")
plt.show()
```

Scatter Plot Output:



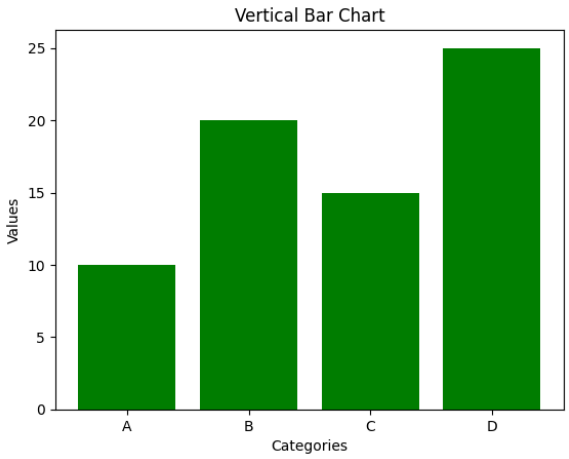
4.3 Bar Charts (Vertical & Horizontal)

Use Case:

- Ideal for comparing categorical data.

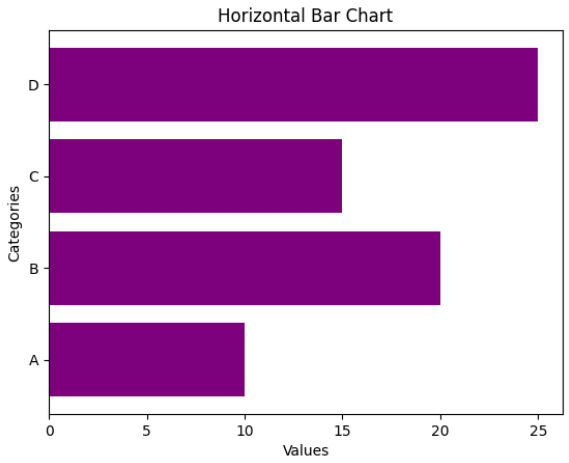
Vertical Bar Chart

```
categories = ["A", "B", "C", "D"]
values = [10, 20, 15, 25]

plt.bar(categories, values, color="green")
plt.xlabel("Categories")
plt.ylabel("Values")
plt.title("Vertical Bar Chart")
plt.show()
```



Horizontal Bar Chart

```
plt.barh(categories, values, color="purple")
plt.xlabel("Values")
plt.ylabel("Categories")
plt.title("Horizontal Bar Chart")
plt.show()
```



Bar Chart Output:

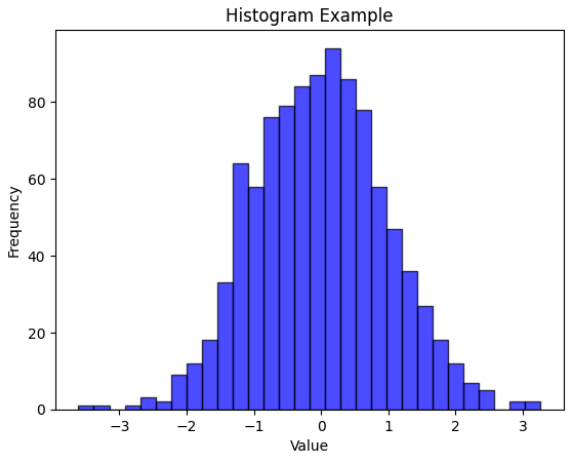
4.4 Histograms & KDE Plots

Use Case:

- Shows the distribution of numerical data.

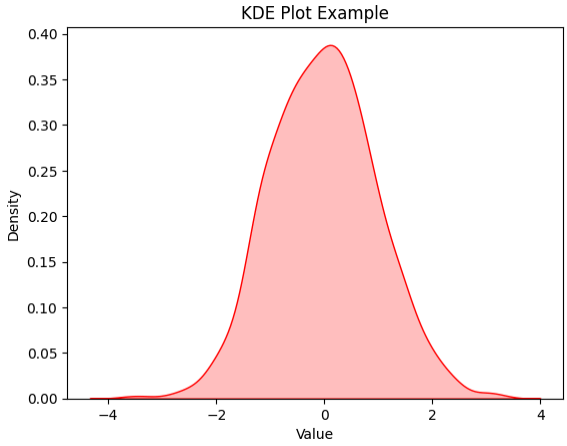
Histogram

```
data = np.random.randn(1000)

plt.hist(data, bins=30, color="blue", alpha=0.7, edgecolor="black")
plt.xlabel("Value")
plt.ylabel("Frequency")
plt.title("Histogram Example")
plt.show()
```



Kernel Density Estimation (KDE) Plot

```
import seaborn as sns

sns.kdeplot(data, shade=True, color="red")
plt.xlabel("Value")
plt.ylabel("Density")
plt.title("KDE Plot Example")
plt.show()
```



Histogram & KDE Plot Output:

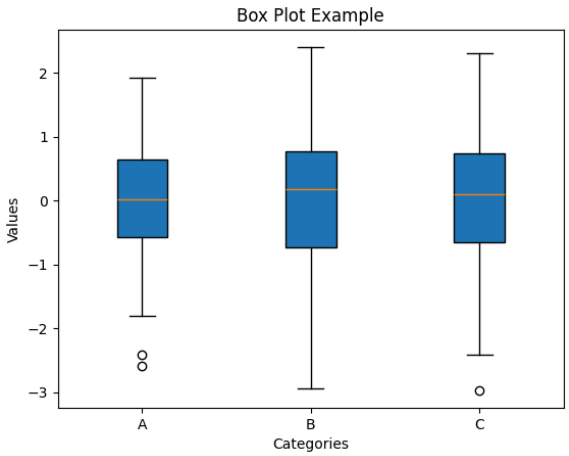
4.5 Box Plots & Violin Plots

Use Case:

- **Box Plots:** Show distribution, outliers, and quartiles.
- **Violin Plots:** Show density in addition to quartiles.

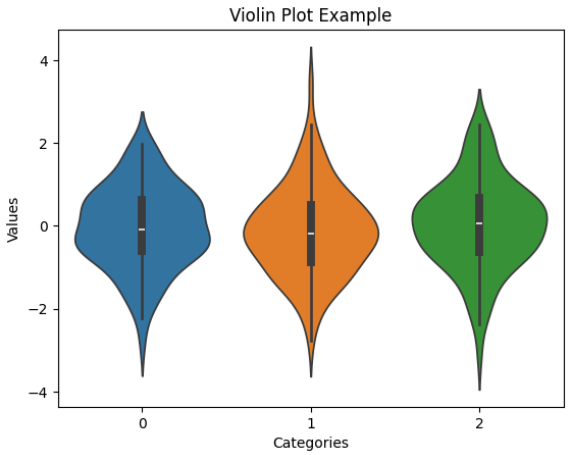
Box Plot

```
data = [np.random.randn(100) for _ in range(3)]

plt.boxplot(data, patch_artist=True, vert=True, labels=["A", "B", "C"])
plt.xlabel("Categories")
plt.ylabel("Values")
plt.title("Box Plot Example")
plt.show()
```



Violin Plot

```
sns.violinplot(data=data)
plt.xlabel("Categories")
plt.ylabel("Values")
plt.title("Violin Plot Example")
plt.show()
```



Box Plot & Violin Plot Output:

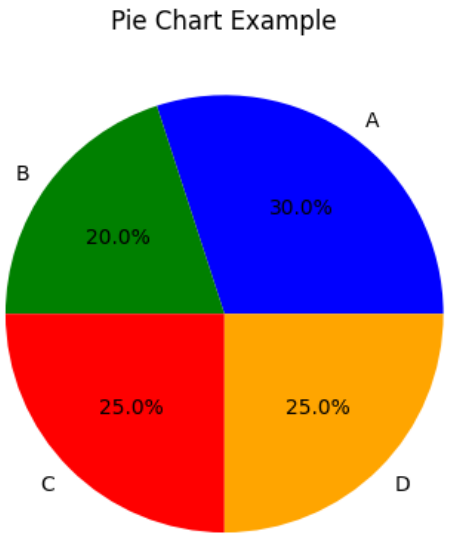
4.6 Pie Charts & Donut Charts

Use Case:

- Used for **percentage-based comparisons**.

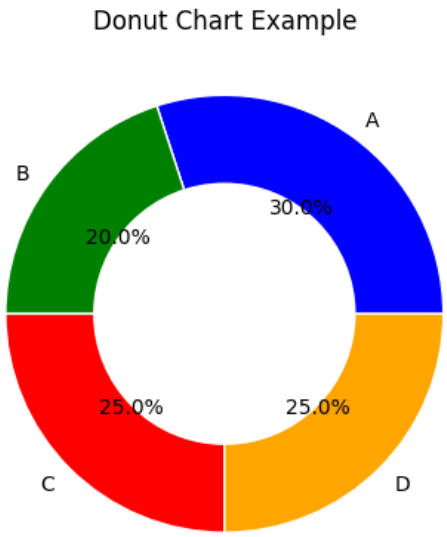
Pie Chart

```
labels = ["A", "B", "C", "D"]
sizes = [30, 20, 25, 25]

plt.pie(sizes, labels=labels, autopct="%1.1f%%", colors=["blue", "green", "red", "orange"])
plt.title("Pie Chart Example")
plt.show()
```



Donut Chart (Modified Pie Chart)

```
plt.pie(sizes, labels=labels, autopct="%1.1f%%", colors=["blue", "green", "red", "orange"], wedgeprops={"edgecolor": "white"})
plt.gca().add_artist(plt.Circle((0,0),0.6,fc='white'))  # Adding a circle in the middle
plt.title("Donut Chart Example")
plt.show()
```



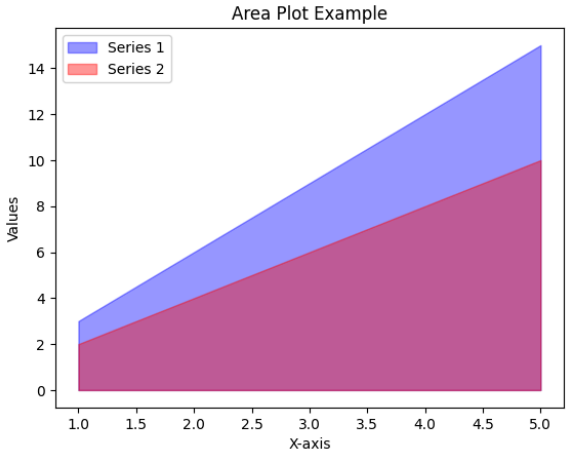
4.7 Area Plots & Stack Plots

Use Case:

- Show **cumulative values** across categories or time.

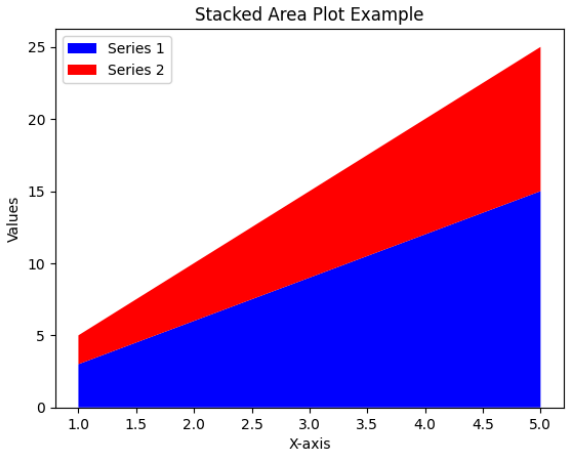
Area Plot

```
x = np.arange(1, 6)
y1 = np.array([3, 6, 9, 12, 15])
y2 = np.array([2, 4, 6, 8, 10])

plt.fill_between(x, y1, color="blue", alpha=0.4, label="Series 1")
plt.fill_between(x, y2, color="red", alpha=0.4, label="Series 2")
plt.xlabel("X-axis")
plt.ylabel("Values")
plt.title("Area Plot Example")
plt.legend()
plt.show()
```



Stacked Plot

```
plt.stackplot(x, y1, y2, colors=["blue", "red"], labels=["Series 1", "Series 2"])
plt.xlabel("X-axis")
plt.ylabel("Values")
plt.title("Stacked Area Plot Example")
plt.legend()
plt.show()
```



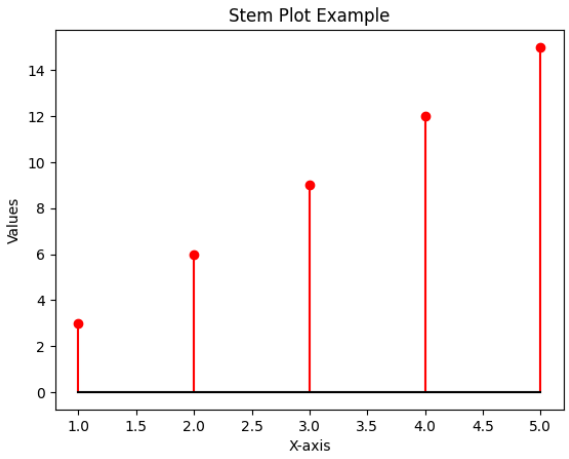
4.8 Stem & Step Plots

Use Case:

- **Stem Plots:** Show individual data points, often used for **discrete signals**.
- **Step Plots:** Show changes in a stepwise fashion, useful for **discrete events**.

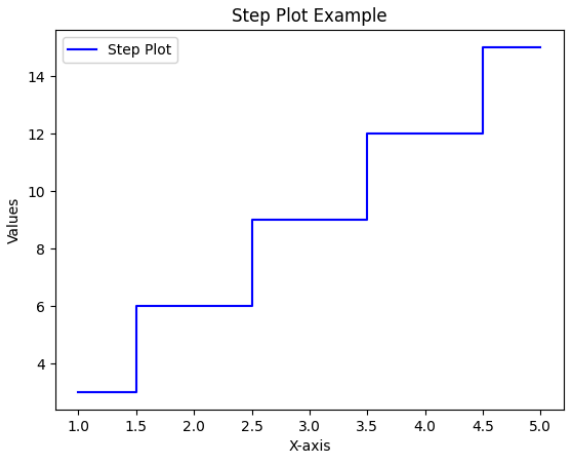
Stem Plot

```
plt.stem(x, y1, linefmt="r-", markerfmt="ro", basefmt="k-")
plt.xlabel("X-axis")
plt.ylabel("Values")
plt.title("Stem Plot Example")
plt.show()
```



Step Plot

```
plt.step(x, y1, where="mid", color="blue", label="Step Plot")
plt.xlabel("X-axis")
plt.ylabel("Values")
plt.title("Step Plot Example")
plt.legend()
plt.show()
```



V. Working with Multiple Plots

Matplotlib provides multiple ways to create **multi-panel visualizations** to compare datasets effectively. The most commonly used approaches include:

1. `plt.subplot()` – Simple subplot creation.
2. `plt.subplots()` – Flexible figure and axes handling.
3. `GridSpec` & `subplot2grid()` – Advanced layout control.

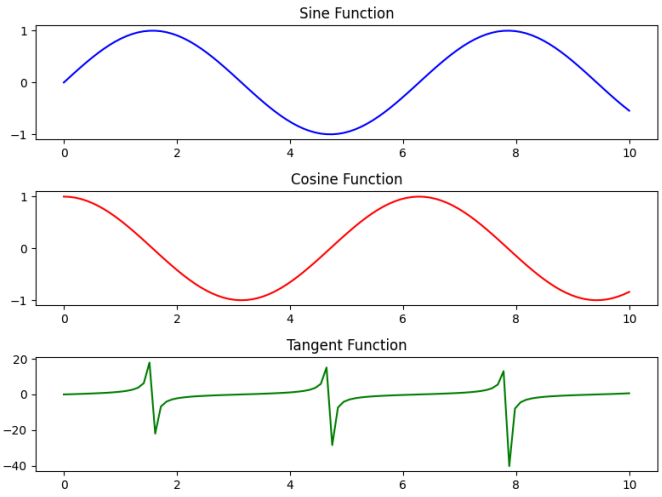
5.1 Creating Multiple Subplots (`plt.subplot` vs. `plt.subplots`)

Using `plt.subplot()` (Single Figure, Indexed Subplots)

- `plt.subplot(nrows, ncols, index)` : Creates a **grid of plots** with `index` referring to the current subplot position.

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

x = np.linspace(0, 10, 100)
```

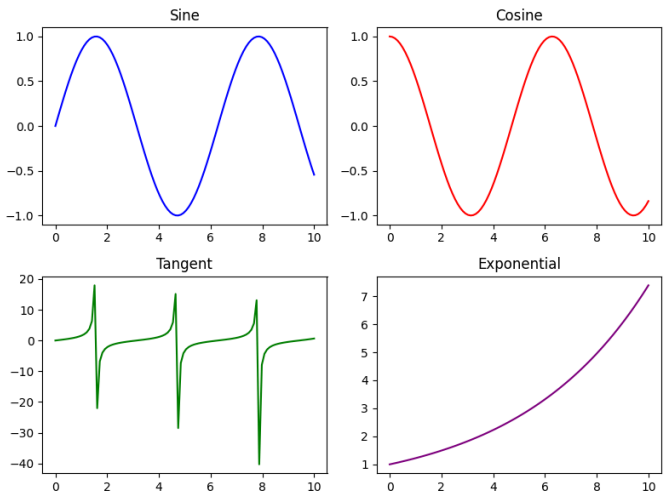
```
y1, y2, y3 = np.sin(x), np.cos(x), np.tan(x)

plt.figure(figsize=(8, 6))

plt.subplot(3, 1, 1)  # 3 rows, 1 column, 1st subplot
plt.plot(x, y1, color="blue")
plt.title("Sine Function")

plt.subplot(3, 1, 2)  # 3 rows, 1 column, 2nd subplot
plt.plot(x, y2, color="red")
plt.title("Cosine Function")

plt.subplot(3, 1, 3)  # 3 rows, 1 column, 3rd subplot
plt.plot(x, y3, color="green")
plt.title("Tangent Function")

plt.tight_layout()  # Adjust spacing
plt.show()
```



Using `plt.subplots()` (More Control, Returns Figure & Axes)

- `plt.subplots(nrows, ncols)`: Returns a **figure object** and an **array of axes**.

```
fig, axes = plt.subplots(2, 2, figsize=(8, 6))

x = np.linspace(0, 10, 100)
axes[0, 0].plot(x, np.sin(x), color="blue")
axes[0, 0].set_title("Sine")

axes[0, 1].plot(x, np.cos(x), color="red")
axes[0, 1].set_title("Cosine")

axes[1, 0].plot(x, np.tan(x), color="green")
axes[1, 0].set_title("Tangent")

axes[1, 1].plot(x, np.exp(x/5), color="purple")
axes[1, 1].set_title("Exponential")

plt.tight_layout()  # Adjusts spacing
plt.show()
```

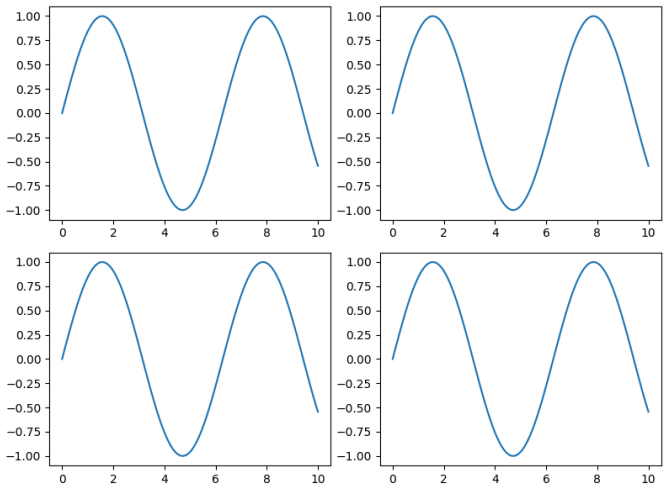


5.2 Adjusting Spacing Between Subplots

When creating multiple subplots, the default layout might **overlap**. Use `plt.tight_layout()` or `fig.subplots_adjust()` to fix spacing.

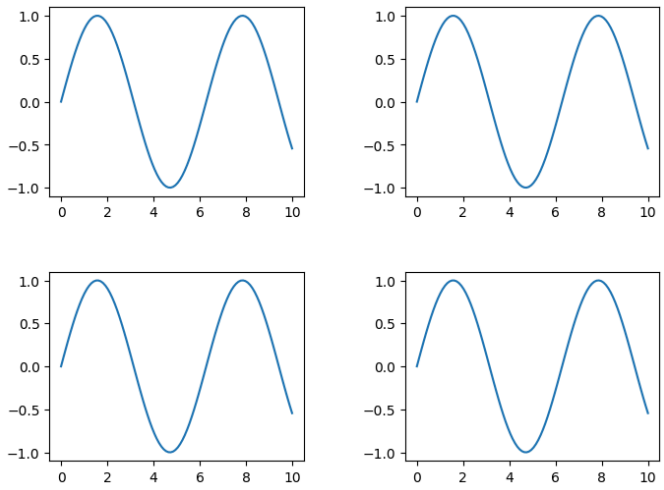
Using `plt.tight_layout()` (Automatic Adjustment)

```
fig, axes = plt.subplots(2, 2, figsize=(8, 6))
for ax in axes.flat:
    ax.plot(x, np.sin(x))
plt.tight_layout()
plt.show()
```



Using `subplots_adjust()` (Manual Control)

```
fig, axes = plt.subplots(2, 2, figsize=(8, 6))
plt.subplots_adjust(wspace=0.4, hspace=0.4)  # Adjust horizontal and vertical space
plt.show()
```

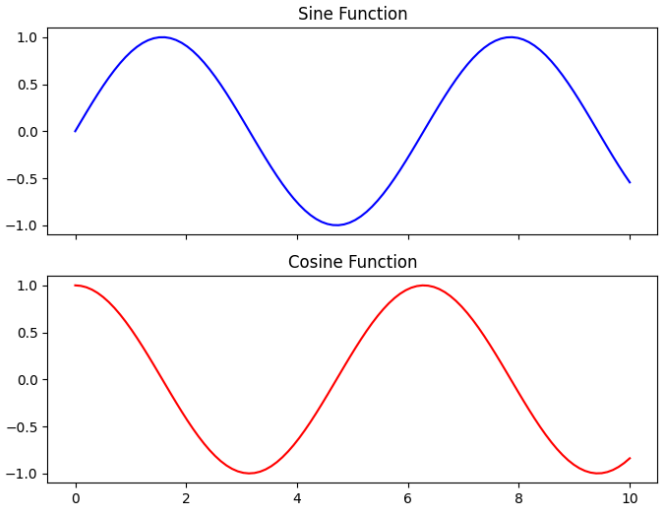


5.3 Sharing Axes Across Subplots

When plotting related data, it is often useful to **share X or Y axes**.

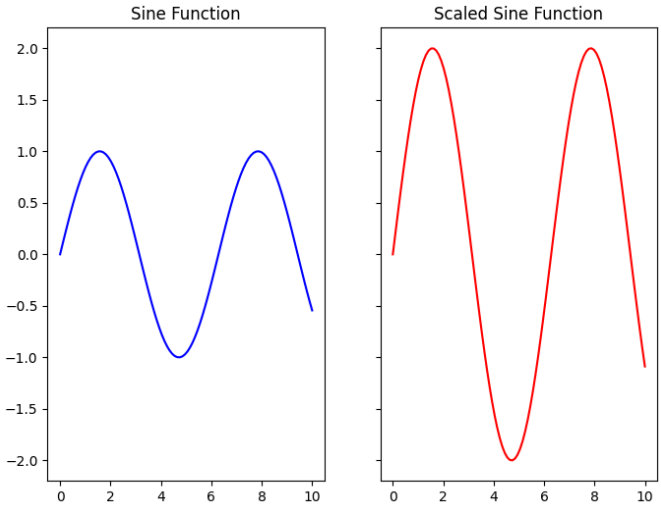
Sharing X-Axis

```
fig, axes = plt.subplots(2, 1, sharex=True, figsize=(8, 6))
axes[0].plot(x, np.sin(x), color="blue")
axes[0].set_title("Sine Function")

axes[1].plot(x, np.cos(x), color="red")
axes[1].set_title("Cosine Function")

plt.show()
```



Sharing Y-Axis

```
fig, axes = plt.subplots(1, 2, sharey=True, figsize=(8, 6))
axes[0].plot(x, np.sin(x), color="blue")
axes[0].set_title("Sine Function")

axes[1].plot(x, np.sin(x) * 2, color="red")
axes[1].set_title("Scaled Sine Function")

plt.show()
```

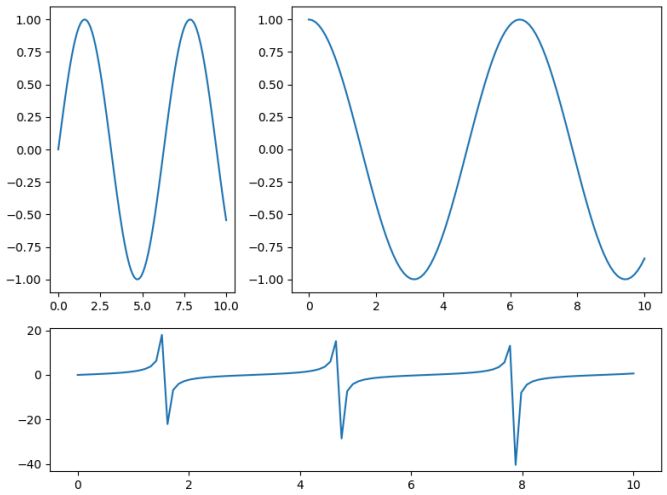


5.4 Different Layouts (GridSpec & `subplot2grid`)

Using `GridSpec` (Advanced Grid Control)

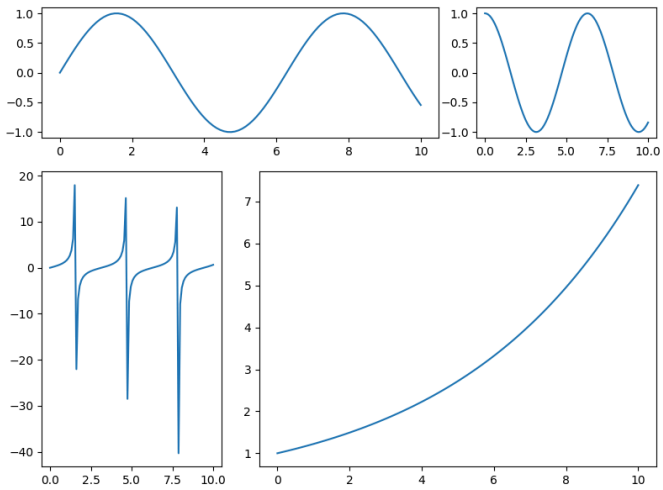
- Allows creating **asymmetrical** or **custom-sized** subplots.

```
import matplotlib.gridspec as gridspec

fig = plt.figure(figsize=(8, 6))
gs = gridspec.GridSpec(2, 2, width_ratios=[1, 2], height_ratios=[2, 1])

ax1 = plt.subplot(gs[0, 0])  # Row 0, Col 0
ax2 = plt.subplot(gs[0, 1])  # Row 0, Col 1
ax3 = plt.subplot(gs[1, :])  # Row 1, spans both columns

ax1.plot(x, np.sin(x))
ax2.plot(x, np.cos(x))
ax3.plot(x, np.tan(x))

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



Using `subplot2grid()` (Manually Place Subplots)

```
fig = plt.figure(figsize=(8, 6))

ax1 = plt.subplot2grid((3, 3), (0, 0), colspan=2)  # Top row, spans 2 columns
ax2 = plt.subplot2grid((3, 3), (0, 2))  # Top right
ax3 = plt.subplot2grid((3, 3), (1, 0), rowspan=2)  # Bottom left, spans 2 rows
ax4 = plt.subplot2grid((3, 3), (1, 1), colspan=2, rowspan=2)  # Bottom right

ax1.plot(x, np.sin(x))
ax2.plot(x, np.cos(x))
ax3.plot(x, np.tan(x))
ax4.plot(x, np.exp(x / 5))

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



VI. Advanced Plot Customization

Customizing plots is essential for **clear, informative, and visually appealing** data visualizations. This section covers advanced customization techniques like

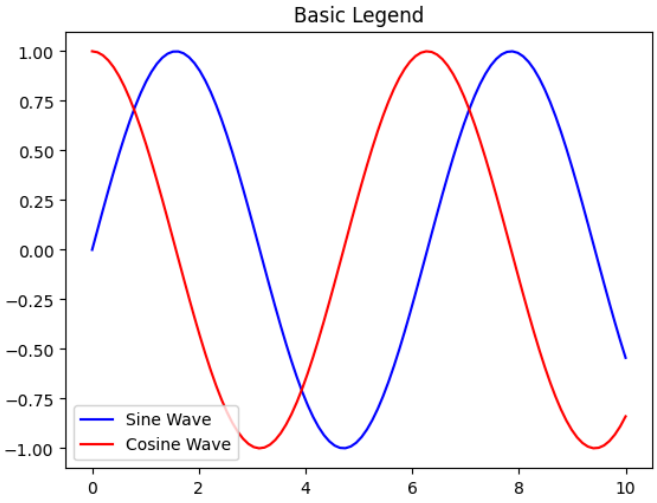
modifying legends, adding arrows, customizing fonts, working with date/time data, and changing themes.

6.1 Customizing Legends

A well-positioned and formatted legend makes a plot more readable. Matplotlib provides extensive customization options using `plt.legend()`.

Basic Legend

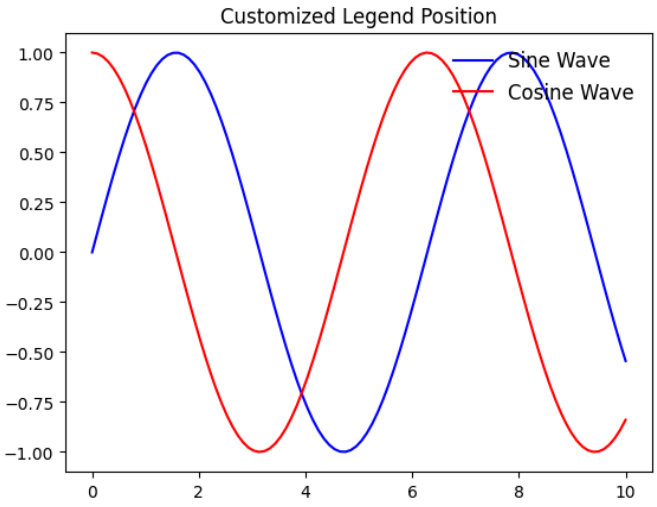
```
import matplotlib.pyplot as plt
import numpy as np

x = np.linspace(0, 10, 100)
y1, y2 = np.sin(x), np.cos(x)

plt.plot(x, y1, label="Sine Wave", color="blue")
plt.plot(x, y2, label="Cosine Wave", color="red")

plt.legend()  # Default placement
plt.title("Basic Legend")
plt.show()
```



Customizing Legend Position

- `loc` **parameter** controls placement (e.g., `'upper left'`, `'lower right'`).
- `bbox_to_anchor` fine-tunes the position.
- `frameon=False` removes the legend box.

```
plt.plot(x, y1, label="Sine Wave", color="blue")
plt.plot(x, y2, label="Cosine Wave", color="red")

plt.legend(loc="upper right", bbox_to_anchor=(1, 1), fontsize=12, frameon=Fal
se)
plt.title("Customized Legend Position")
plt.show()
```



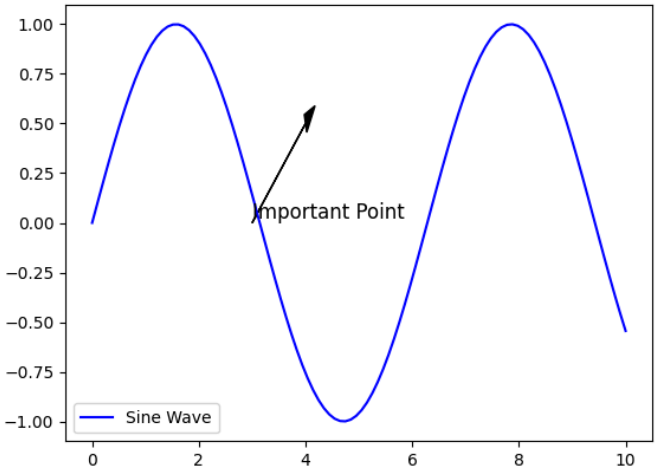
6.2 Adding Arrows & Shapes

Annotations such as arrows and shapes help highlight important points.

Adding Arrows (`plt.arrow()`)

```
plt.plot(x, y1, label="Sine Wave", color="blue")
plt.arrow(3, 0, 1, 0.5, head_width=0.1, head_length=0.2, fc="black", ec="blac
```

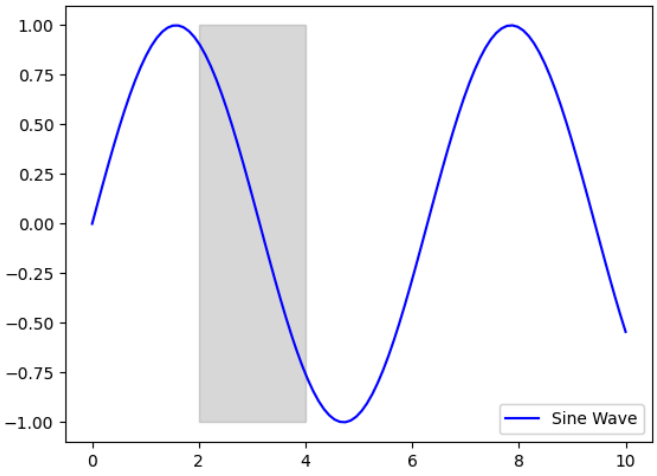
```
k")

plt.text(3, 0, "Important Point", fontsize=12, verticalalignment="bottom")
plt.legend()
plt.show()
```



Adding Rectangles, Circles, and Other Shapes

Using `patches` from Matplotlib:

```
import matplotlib.patches as patches

fig, ax = plt.subplots()
ax.plot(x, y1, label="Sine Wave", color="blue")

# Adding a rectangle
rect = patches.Rectangle((2, -1), 2, 2, color="gray", alpha=0.3)
ax.add_patch(rect)

plt.legend()
plt.show()
```

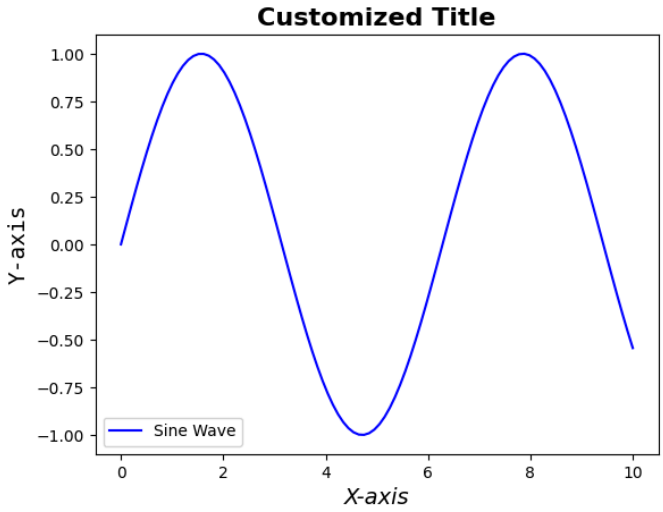


6.3 Customizing Fonts & Styles

Changing Font Styles

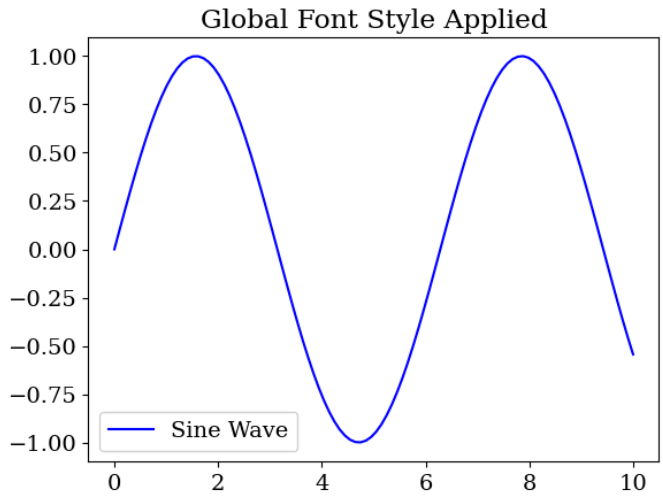
You can customize font properties globally using `rcParams` or locally using `fontdict`.

```
plt.plot(x, y1, label="Sine Wave", color="blue")

plt.title("Customized Title", fontsize=16, fontweight="bold", fontname="Arial")
plt.xlabel("X-axis", fontsize=14, fontstyle="italic")
plt.ylabel("Y-axis", fontsize=14, fontfamily="monospace")

plt.legend()
plt.show()
```



Setting Global Font Styles

```
plt.rcParams["font.size"] = 14
plt.rcParams["font.family"] = "serif"

plt.plot(x, y1, label="Sine Wave", color="blue")
plt.title("Global Font Style Applied")
plt.legend()
plt.show()
```

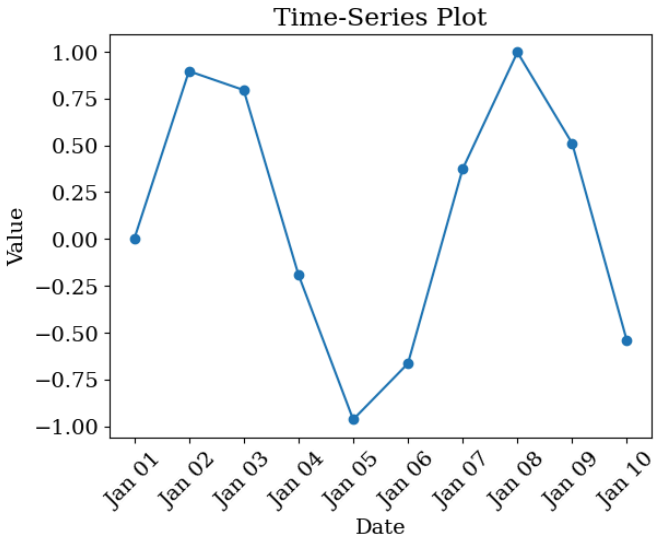


6.4 Working with Date & Time Data in Plots

Matplotlib supports time-series data using `matplotlib.dates` and `datetime`.

Plotting Time-Series Data

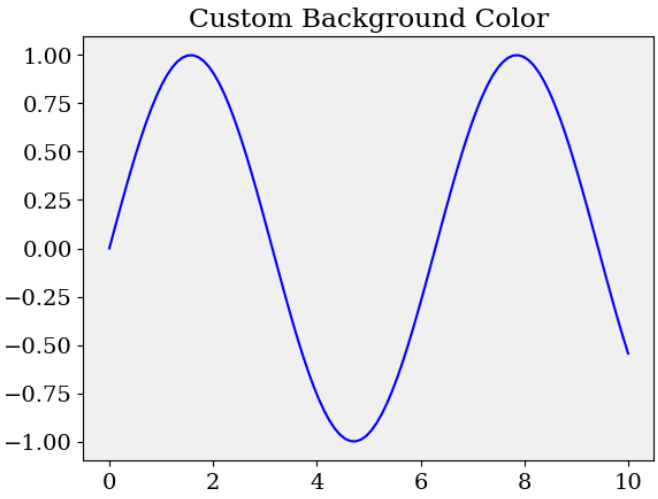
```
import matplotlib.dates as mdates
import datetime

# Generate sample date-time data
dates = [datetime.date(2023, 1, i) for i in range(1, 11)]
values = np.sin(np.linspace(0, 10, 10))

fig, ax = plt.subplots()
ax.plot(dates, values, marker="o", linestyle="-")

ax.xaxis.set_major_formatter(mdates.DateFormatter("%b %d"))  # Format dat
es
ax.xaxis.set_major_locator(mdates.DayLocator(interval=1))  # Show every day

plt.title("Time-Series Plot")
plt.xlabel("Date")
plt.ylabel("Value")
plt.xticks(rotation=45)
plt.show()
```



6.5 Customizing Backgrounds & Themes

Changing Background Color

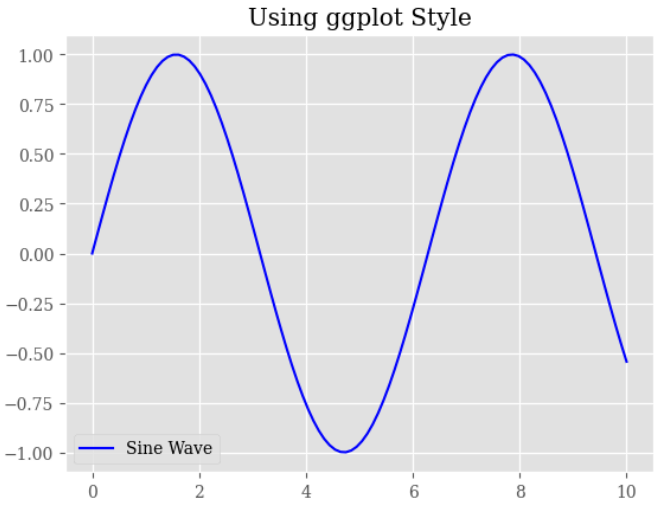
```
fig, ax = plt.subplots()
ax.plot(x, y1, label="Sine Wave", color="blue")

ax.set_facecolor("#f0f0f0")  # Light gray background
plt.title("Custom Background Color")
plt.show()
```



Using Built-in Styles

Matplotlib provides several predefined styles. Use `plt.style.available` to list them.

```
import matplotlib.style as style

plt.style.use("ggplot")  # Apply ggplot style

plt.plot(x, y1, label="Sine Wave", color="blue")
plt.title("Using ggplot Style")
plt.legend()
plt.show()
```



Creating a Custom Theme

```
plt.rcParams["axes.facecolor"] = "#f5f5f5"
plt.rcParams["axes.edgecolor"] = "#333333"
plt.rcParams["grid.color"] = "gray"

plt.plot(x, y1, label="Sine Wave", color="blue")
plt.title("Custom Theme Applied")
plt.legend()
plt.show()
```