LEARN COMPLETE

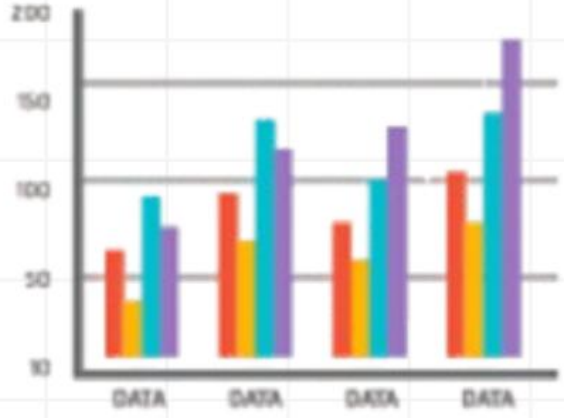
MATPLOTLIB

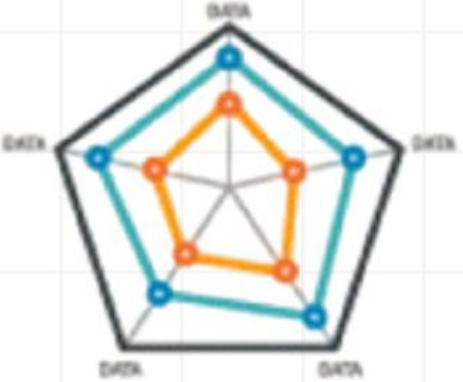
WITH MOST ASKED INTERVIEW QUESTIONS

CURATED BY

BHAVESH ARORA
SAURABH G

matplotlib





Crack Top Analyst Roles with This 20-Day MATPLOTLIB Series!

Learn Complete MATPLOTLIB with Most ASKED INTERVIEW Questions

Are you aiming for roles like Business Analyst, Data Analyst, Power BI Developer, or BI Consultant at top companies?

Curated by

SAURABH G
Founder at DataNiti
6+ Years of Experience | Senior Data Engineer
Linkedin: www.linkedin.com/in/saurabhgghatnekar

BHAVESH ARORA
Senior Data Analyst at Delight Learning Services
M.Tech – IIT Jodhpur | 3+ Years of Experience
Linkedin: www.linkedin.com/in/bhavesh-arora-11b0a319b

Connect with us: https://topmate.io/bhavesh_arora/

Let's embark on this journey together and make your dreams a reality, starting today.

Day 1: Introduction to Matplotlib and Basic Line Plot

Why Matplotlib?

- **Most powerful** library for creating visualizations in Python.
- Helps to **analyze patterns** and **present insights** visually.
- Works well with **NumPy**, **Pandas**, and **SciPy**.

---

1. Importing Matplotlib

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

- pyplot is a sub-library of Matplotlib — designed like MATLAB's plotting interface.

---

2. Basic Line Plot

```
x = [1, 2, 3, 4, 5]
y = [2, 3, 5, 7, 11]

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

- **plot(x, y)**: Draws a line connecting (x, y) points.
- **show()**: Renders the plot window.

---

3. Anatomy of a Figure

In Matplotlib, a figure contains:

- **Figure** → Full canvas.
- **Axes** → Plot area inside the figure (can have multiple).
- **Axis** → X-axis and Y-axis inside an Axes.

```
fig, ax = plt.subplots()
ax.plot(x, y)
plt.show()
```

- fig = Entire figure object.
- ax = Single subplot (Axes) where data is plotted.

---

4. Save a Plot

```
plt.plot(x, y)
plt.savefig('line_plot.png')
plt.show()
```

- savefig('filename.png'): Saves the current figure to a file.

Key Takeaways:

- **Import** pyplot for easy plotting.
- Use **plot()** to draw simple graphs.
- **Figure** → **Axes** → **Axis** is the structure.
- Always use **show()** to display or **savefig()** to save.

Interview Questions (Medium to Hard):

Q1: How does Matplotlib's object-oriented API (using Figure and Axes) differ from the pyplot state-machine API?
A:

- The pyplot API (plt.plot()) is like a state machine: it keeps track of the current figure and axes.
- The object-oriented API (fig, ax = plt.subplots()) gives explicit control over each figure and axes, allowing multiple customizations, better for large/complex plots.

Q2: In a Matplotlib figure, how would you create two subplots and plot different datasets without using plt.subplot()?
A:
Use fig, (ax1, ax2) = plt.subplots(1, 2) and then use ax1.plot(...), ax2.plot(...) separately. This is more maintainable for multiple subplots.

Q3: If you call plt.plot() multiple times without calling plt.show(), what happens?
A:
All plots get layered on the same figure until show() is called. Each plot() adds another line unless a new figure is created with plt.figure() or fig, ax = plt.subplots().

✅ **End of Day 1!**

Day 2: Customizing Line Styles, Colors, and Markers

Why Customize Plots?

- **Enhance readability** and **aesthetics**.
- **Highlight trends** or **differentiate** multiple lines.
- **Match** publication or brand styles.

---

1. Change Line Color

```
x = [1, 2, 3, 4]
y = [2, 4, 6, 8]

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

- color='color_name' → Changes line color (e.g., 'red', 'blue', HEX codes like #FF5733).

---

2. Change Line Style

```
plt.plot(x, y, linestyle='--')  # Dashed line
plt.show()
```

- Common options:
  - '-' → Solid
  - '--' → Dashed
  - '-.' → Dash-dot
  - ':' → Dotted

---

3. Add Markers

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

- marker='o' → Circle markers at data points.
- Other markers:
  - 's' → Square
  - '^' → Triangle Up
  - '*' → Star

---

4. Combine All Customizations

```
plt.plot(x, y, color='purple', linestyle='-.', marker='s', linewidth=2, markersize=8)
plt.show()
```

- **linewidth**: Thickness of the line.
- **markersize**: Size of the marker points.

---

5. Short-Hand Notation

```
plt.plot(x, y, 'g--o')
plt.show()
```

- 'g' → Green
- '--' → Dashed Line
- 'o' → Circle Marker

Key Takeaways:

- Customize plots for **clarity** and **professionalism**.
- Control **color**, **style**, and **markers** individually or using **short-hand**.
- linewidth and markersize add further control.

Interview Questions (Medium to Hard):

Q1: What is the precedence when you specify both shorthand and explicit arguments like color, linestyle, and marker?
A:

- Explicit keyword arguments (like color='red', linestyle='--') **override** shorthand style strings if conflicts arise.

Q2: How would you plot multiple lines with different styles efficiently inside a loop?
A:

- Maintain lists of colors, markers, linestyles, and loop through them when plotting. Example:

```
styles = ['r--o', 'b-.s', 'g:*']
for i in range(3):
    plt.plot(x, [v * (i+1) for v in y], styles[i])
```

Q3: How would you customize a plot when color-blind users must be able to distinguish the lines?

A:

- Use **different markers** and **different linestyles**, not just colors.
- Prefer **color palettes** like Color Universal Design (CUD) or use **Matplotlib's color-blind friendly colormaps** like 'tab10', 'Set2', etc.

✅ **End of Day 2!**

Day 3: Adding Titles, Labels, and Legends

Why Add Titles and Labels?

- Make plots **self-explanatory** without additional description.
- Improve **interpretability** and **professional appearance**.
- **Identify** multiple lines or datasets clearly with legends.

---

1. Adding Title, X-axis, and Y-axis Labels

```
x = [1, 2, 3, 4]
y = [2, 4, 6, 8]

plt.plot(x, y)
plt.title('Simple Line Plot')
plt.xlabel('X-axis Values')
plt.ylabel('Y-axis Values')
plt.show()
```

- title(): Adds a title to the plot.
- xlabel() and ylabel(): Add labels to the axes.

---

2. Customize Titles and Labels

```
plt.plot(x, y)
plt.title('Customized Plot', fontsize=16, color='blue')
plt.xlabel('Input', fontsize=12)
plt.ylabel('Output', fontsize=12)
plt.show()
```

- Customize **fontsize**, **fontweight**, **color**, etc.

---

3. Adding a Legend

```
plt.plot(x, y, label='y = 2x')
plt.legend()
plt.show()
```

- label='...': Set label while plotting.
- legend(): Displays the label in a box.

---

4. Set Legend Location

```
plt.plot(x, y, label='y = 2x')
plt.legend(loc='upper left')
plt.show()
```

- Common loc values:
  - 'upper right' (default)
  - 'upper left'
  - 'lower right'
  - 'lower left'
  - 'best': Auto-pick best location.

---

5. Multiple Lines with Legends

```
y2 = [3, 6, 9, 12]

plt.plot(x, y, label='2x')
plt.plot(x, y2, label='3x')
plt.title('Multiple Lines')
plt.xlabel('X Values')
plt.ylabel('Y Values')
plt.legend()
plt.show()
```

- Each line can have its **own label**.

Key Takeaways:

- Use **titles** and **axis labels** to explain plots clearly.
- Always add a **legend** when plotting multiple datasets.
- Customize font size, color, and location to match style needs.

Interview Questions (Medium to Hard):

Q1: How does plt.legend() decide the placement when loc='best' is used, and what can go wrong with it?
A:

- 'best' automatically finds an area with the least data overlap.
- However, if the plot is densely packed, even 'best' can overlap with critical parts of the graph, requiring manual adjustment.

Q2: How would you create a custom legend without associating it directly with plotted lines?
A:

- Use plt.legend(handles=[custom_lines], labels=['label1', 'label2']) with custom Line2D objects from matplotlib.lines. Example:

```
from matplotlib.lines import Line2D
custom = [Line2D([0], [0], color='blue', lw=2),
     Line2D([0], [0], color='green', lw=2, linestyle='--')]
     plt.legend(custom, ['2x', '3x'])
```

Q3: Why should you avoid hardcoding legend locations in dynamic plots, and what's a better strategy?
A:

- Hardcoding (loc='upper right') may cause overlap when plot contents change.
- A better strategy is dynamic calculation using **bounding boxes** (bbox_to_anchor) or designing plots with margins to accommodate legends.

✅ **End of Day 3!**

Day 4: Changing Figure Size, DPI, and Saving Images

Why Customize Size & DPI?

- Improve **readability** for presentations or reports
- Make plots **publication-ready** or web-optimized
- Save **space** in dashboards or dense plots

---

1. Changing Figure Size with plt.figure(figsize=(w, h))

```
import matplotlib.pyplot as plt

x = [1, 2, 3, 4]
y = [2, 4, 6, 8]
```

```
plt.figure(figsize=(8, 4))  # Width=8 inches, Height=4 inches
plt.plot(x, y)
plt.title('Custom Figure Size')
plt.show()
```

- figsize: Tuple (width, height) in inches
- Good for balancing screen space and label readability

---

2. Changing DPI (Dots Per Inch)

```
plt.figure(figsize=(6, 3), dpi=150)
plt.plot(x, y)
plt.title('Higher DPI')
plt.show()
```

- Higher dpi = **sharper images**
- Useful for print-quality images (e.g., 300 DPI for journals)

---

3. Saving the Plot as a File

```
plt.plot(x, y)
plt.title('Saving Example')
plt.savefig('myplot.png')  # Default DPI = 100
```

- Saves the current figure to disk
- File formats supported: .png, .jpg, .svg, .pdf, etc.

---

4. Save with Custom DPI & Transparent Background

```
plt.plot(x, y)
plt.title('High Quality Transparent')
plt.savefig('clean_plot.png', dpi=300, transparent=True)
```

- transparent=True: Saves without white background
- Perfect for overlays in presentations or web graphics

---

5. Save Without Extra Padding

```
plt.plot(x, y)
plt.title('Tight Layout')
plt.tight_layout()
plt.savefig('tight_plot.png', bbox_inches='tight')
```

- bbox_inches='tight': Removes unwanted whitespace
- tight_layout(): Adjusts padding between elements

Key Takeaways:

- Use figsize to control plot space visually.
- Use high dpi for clear output in reports or printing.
- savefig() offers options for size, quality, transparency.
- Combine tight_layout() with bbox_inches='tight' to avoid cutoff or padding issues.

Interview Questions (Medium to Hard):

Q1: You notice labels are cut off in your saved image. How would you fix this using matplotlib?

A: Use plt.tight_layout() before savefig() and add bbox_inches='tight' to avoid clipping.

Example:
plt.tight_layout()
plt.savefig('plot.png', bbox_inches='tight')

Q2: How does changing DPI affect file size and resolution in saved figures?
A:

- DPI controls how many pixels are rendered per inch.
- Higher DPI = larger file size and sharper image.
- Critical for print vs. screen distinction. E.g., 72–100 for web, 300+ for publications.

Q3: What is the difference between figsize and dpi, and how do they combine to define image resolution?
A:

- figsize defines size in **inches**, dpi defines **pixels per inch**.
- Total resolution = figsize[0] * dpi (width in px) × figsize[1] * dpi (height in px).
- For example: figsize=(8, 4), dpi=100 → image is 800×400 px.

Q4: Why might you choose transparent=True when saving a plot, and what are the risks?
A:

- Use transparent=True when overlaying plots on non-white backgrounds (e.g., websites or slides).
- **Risk**: If plot elements (e.g., text) are also light-colored, visibility might be reduced or lost.

Day 5: Subplots — One Figure, Multiple Plots in Matplotlib

Why Use Subplots?

- Compare multiple charts side by side
- Show different dimensions of the same data
- Keep your visualizations **organized and clean**

---

1. Basic Subplots with plt.subplot()

```
import matplotlib.pyplot as plt

x = [1, 2, 3, 4]
y1 = [1, 4, 9, 16]
y2 = [2, 3, 5, 7]

plt.subplot(1, 2, 1)  # (rows, cols, index)
plt.plot(x, y1)
plt.title('Square')

plt.subplot(1, 2, 2)
plt.plot(x, y2)
plt.title('Prime')

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

- 1 row, 2 columns
- Index starts at 1, not 0
- Use tight_layout() to avoid overlap

---

2. Using plt.subplots() — Cleaner and Scalable

```
fig, axs = plt.subplots(1, 2, figsize=(8, 4))  # 1 row, 2 columns
axs[0].plot(x, y1)
axs[0].set_title('Square')

axs[1].plot(x, y2)
axs[1].set_title('Prime')

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

- axs: Array of Axes objects (can be 1D or 2D)
- Preferred for cleaner, flexible code

---

3. Looping Through Multiple Subplots

```
y_data = [[1, 4, 9], [1, 2, 3], [3, 2, 1]]
titles = ['Square', 'Linear', 'Reverse']

fig, axs = plt.subplots(1, 3, figsize=(10, 3))

for i in range(3):
    axs[i].plot(y_data[i])
    axs[i].set_title(titles[i])

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

- Efficient when plotting many charts
- Makes code DRY (Don't Repeat Yourself)

---

4. Sharing Axis

```
fig, axs = plt.subplots(2, 1, sharex=True)
axs[0].plot([1, 2, 3], [2, 3, 4])
axs[1].plot([1, 2, 3], [4, 1, 3])
plt.show()
```

- sharex=True, sharey=True: Sync axes for comparison
- Useful in time-series or grouped data plots

---

5. Grid of Subplots (2D)

```
fig, axs = plt.subplots(2, 2, figsize=(8, 6))
axs[0, 0].plot([1, 2, 3], [1, 4, 9])
axs[0, 1].bar([1, 2, 3], [3, 2, 1])
axs[1, 0].scatter([1, 2, 3], [9, 6, 3])
axs[1, 1].hist([1, 2, 2, 3], bins=3)
plt.tight_layout()
plt.show()
```

- 2×2 layout: Ideal for mini dashboards

Key Takeaways:

- subplot() for quick demos, subplots() for real work
- Looping over axes helps scale
- Always use tight_layout()
- Share axes when comparing patterns

Interview Questions (Medium to Hard):

Q1: What's the difference between plt.subplot() and plt.subplots()?
A:

- subplot() creates a single plot within a figure using row-col-index structure.
- subplots() creates a grid and returns both the Figure and an array of Axes objects, allowing more flexibility and better control.

Q2: You have a 2×2 subplot grid, but only want to display 3 plots. How do you leave one blank without breaking the layout?
A:

- Skip plotting in one of the axes or turn it off:
  axs[1, 1].axis('off')

Q3: Why would you use sharex=True or sharey=True in subplots?
A:

- Ensures consistency across plots.
- Useful for comparing trends without distraction from different scales.
- Reduces visual clutter (less ticks/labels).

Q4: How can you dynamically plot an unknown number of charts using subplots?
A:

- Use plt.subplots(n) where n is the number of charts.
- Loop through the returned axs and plot:

```
fig, axs = plt.subplots(n)
for i, ax in enumerate(axs):
    ax.plot(data[i])
```

✅ **End of Day 5!**

🗓️ Day 6: Customizing Axes, Ticks & Labels

Why Customize Axes & Labels?

- **Improve readability** for the audience
- Provide **context** (e.g., units, timeframes)
- Create **visual clarity** and make data easier to understand

---

1. Customizing Axis Labels with set_xlabel() and set_ylabel()

```
import matplotlib.pyplot as plt

x = [1, 2, 3, 4]
y = [2, 4, 6, 8]

plt.plot(x, y)
plt.title('Custom Axis Labels')
plt.xlabel('Time (s)')
plt.ylabel('Distance (m)')
plt.show()
```

- set_xlabel(): Label for the x-axis
- set_ylabel(): Label for the y-axis
- Adding units or context makes your plots clearer

---

2. Customizing Tick Marks with xticks() and yticks()

```
plt.plot(x, y)
plt.title('Custom Ticks')
plt.xticks([1, 2, 3, 4], ['1s', '2s', '3s', '4s'])  # Custom x-tick labels
plt.yticks([2, 4, 6, 8], ['2 meters', '4 meters', '6 meters', '8 meters'])  # Custom y-tick labels
plt.show()
```

- xticks(): Set custom tick locations and labels for x-axis
- yticks(): Set custom tick locations and labels for y-axis
- You can adjust the position and label content as needed

---

3. Rotating Tick Labels for Better Readability

```
labels = ['Jan', 'Feb', 'Mar', 'Apr', 'May']
values = [10, 15, 7, 10, 20]

plt.plot(labels, values)
plt.xticks(rotation=45)  # Rotates x-axis labels by 45 degrees
plt.show()
```

- Use rotation for **angled labels**, especially useful when the labels overlap

---

4. Customizing Axis Limits with set_xlim() and set_ylim()

```
plt.plot(x, y)
plt.title('Custom Axis Limits')
plt.xlim(0, 5)  # Set x-axis from 0 to 5
```

```
plt.ylim(0, 10)  # Set y-axis from 0 to 10
plt.show()
```

- set_xlim(), set_ylim(): Control the axis limits
- Useful for focusing on specific regions of the data

---

5. Logarithmic Scales

```
plt.plot(x, y)
plt.title('Logarithmic Scale')
plt.xscale('log')  # Set x-axis to logarithmic scale
plt.yscale('log')  # Set y-axis to logarithmic scale
plt.show()
```

- xscale='log', yscale='log': Converts axes to a log scale
- Useful for data spanning several orders of magnitude (e.g., scientific data, financial data)

---

6. Using twinx() to Create Two Y-Axes

```
fig, ax1 = plt.subplots()

ax1.plot(x, y, 'g-')
ax1.set_xlabel('Time (s)')
ax1.set_ylabel('Distance (m)', color='g')

ax2 = ax1.twinx()  # Creates a second y-axis
ax2.plot(x, [i*2 for i in y], 'b-')
ax2.set_ylabel('Speed (m/s)', color='b')

plt.show()
```

- twinx(): Adds a secondary y-axis on the right side
- Useful when plotting two datasets with different scales (e.g., distance vs. speed)

Key Takeaways:

- xlabel(), ylabel() add context to axes.
- xticks(), yticks() customize tick labels for clarity.
- Axis limits and log scales help focus on relevant data.
- twinx() is great for dual axes with different units/scales.

Interview Questions (Medium to Hard):

Q1: How would you adjust the axis limits if you have an outlier in your dataset that makes the rest of the data hard to visualize?
A:

- Use set_xlim() and set_ylim() to zoom in on the relevant portion of the data and exclude outliers.
- Alternatively, you could use a log scale to compress the scale and show the data more effectively.

Q2: What is the advantage of using a logarithmic scale in plots, and when should it be used?
A:

- Logarithmic scales are used when data spans several orders of magnitude. They help visualize data that has exponential growth, like population growth or financial data.
- Example: Plotting the population growth of countries or stock prices over time.

Q3: You're plotting a dataset with categories on the x-axis and values on the y-axis. However, the x-axis labels are overlapping. How would you solve this issue?
A:

- Use plt.xticks(rotation=45) to rotate the labels for better visibility.
- Alternatively, reduce the number of ticks or adjust their positions using xticks().

Q4: You need to create two plots in one figure: one showing distance over time and another showing speed over time. The y-axes are in different units. How would you do this in Matplotlib?
A:

- Use twinx() to create two y-axes: one on the left for distance, another on the right for speed.
  Example:

```
ax1 = plt.gca()  # Get current axes
ax1.plot(x, distance_data)
ax2 = ax1.twinx()
ax2.plot(x, speed_data)
```

✅ **End of Day 6!**

Day 7: Customizing Legends, Titles, and Gridlines

Why Customize Legends, Titles, and Gridlines?

- **Legends**: Help differentiate between multiple data series in a plot.
- **Titles**: Provide context for what the plot represents.
- **Gridlines**: Improve readability, especially when reading off values.

---

1. Adding Titles with plt.title()

```
import matplotlib.pyplot as plt

x = [1, 2, 3, 4]
y = [2, 4, 6, 8]

plt.plot(x, y)
plt.title('Linear Growth Over Time')
plt.xlabel('Time (s)')
plt.ylabel('Distance (m)')
plt.show()
```

- Use plt.title() to add a title to your plot, making it clear what the graph represents.
- You can also customize the font size, family, and style using parameters like fontsize, fontweight, and family.

---

2. Customizing Legends with plt.legend()

```
x = [1, 2, 3, 4]
y1 = [2, 4, 6, 8]
y2 = [1, 3, 5, 7]

plt.plot(x, y1, label='Distance')
plt.plot(x, y2, label='Speed')

plt.xlabel('Time (s)')
plt.ylabel('Value')
plt.legend(title='Measurements')
plt.show()
```

- plt.legend(): Adds a legend to differentiate between multiple data series.
- You can specify labels for each plot using the label parameter inside the plot() function.
- Customize the legend's position using loc='upper left', loc='best', etc.

---

3. Customizing Gridlines with plt.grid()

```
plt.plot(x, y1, label='Distance')
plt.plot(x, y2, label='Speed')

plt.xlabel('Time (s)')
plt.ylabel('Value')
plt.title('Speed vs Distance')

plt.grid(True)  # Enable gridlines
plt.legend()
plt.show()
```

- plt.grid(True): Turns on gridlines to help read values more easily.
- You can control the gridline style using parameters like linestyle='--', linewidth=0.5, and color='gray'.

---

4. Customizing the Gridline Style

```
plt.plot(x, y1, label='Distance')
plt.plot(x, y2, label='Speed')

plt.xlabel('Time (s)')
plt.ylabel('Value')
plt.title('Speed vs Distance')

# Custom gridline style
plt.grid(True, linestyle=':', linewidth=1.5, color='green')
plt.legend()
plt.show()
```

- You can modify the gridline style using linestyle, linewidth, and color for a more customized look.

---

5. Adjusting Plot and Legend Placement

```
plt.plot(x, y1, label='Distance')
plt.plot(x, y2, label='Speed')

plt.xlabel('Time (s)')
plt.ylabel('Value')
plt.title('Speed vs Distance')

plt.legend(loc='best')  # Automatically place legend in the best location
plt.grid(True)
plt.tight_layout()  # Automatically adjust subplots to fit content
plt.show()
```

- loc='best': Automatically places the legend in the optimal position to avoid overlap.
- tight_layout(): Adjusts the spacing of the plot to ensure no labels or titles are clipped.

Key Takeaways:

- **Titles**: Provide context to the plot using plt.title().
- **Legends**: Differentiate multiple data series with plt.legend(). Use loc to position it optimally.
- **Gridlines**: Improve readability using plt.grid(), and customize their style for clarity.

Interview Questions (Medium to Hard):

Q1: How would you add a title to a plot in Matplotlib, and why is it important?
A:

- You can add a title using plt.title('Your Title'). A title helps provide context for the plot, making it clear what the plot represents (e.g., "Sales Growth Over Time").

Q2: You have multiple plots, and the axis labels or legends are being cut off. How would you fix this?
A:

- Use plt.tight_layout() to automatically adjust the spacing between elements in the plot and prevent clipping of labels or titles.
- Alternatively, you can manually adjust the margins using plt.subplots_adjust().

Q3: You're plotting two series, one for distance and one for speed, but the lines overlap in the legend. How would you fix this?
A:

- You can modify the legend's position using the loc parameter, e.g., plt.legend(loc='upper left') to move the legend to a less crowded area.
- Alternatively, use plt.legend(title='Measurements') to add a title to the legend for clarity.

Q4: In what scenarios would you use plt.grid(True) in a plot, and how can you customize the gridlines' appearance?
A:

- plt.grid(True) is useful when you need to make the values in your plot easier to read by providing horizontal and vertical lines.
- You can customize the gridlines by adjusting linestyle, linewidth, and color parameters, e.g., plt.grid(True, linestyle='--', color='gray').

✅ **End of Day 7!**

Day 8: Customizing Colors and Styles in Matplotlib

Why Customize Colors and Styles?

- **Colors**: Differentiate data series and add meaning (e.g., red for negative, green for positive).
- **Styles**: Customize the line, marker, and other visual elements to match the presentation style or make specific data stand out.

---

1. Basic Color Customization

```
import matplotlib.pyplot as plt

x = [1, 2, 3, 4]
y = [2, 4, 6, 8]

plt.plot(x, y, color='purple')  # Set line color to purple
plt.title('Customized Line Color')
plt.show()
```

- You can specify colors using named colors (e.g., 'red', 'green'), RGB tuples (e.g., (0.1, 0.2, 0.5)), or hexadecimal color codes (e.g., '#FF5733').

---

2. Using Color Codes

```
plt.plot(x, y, color='#FF5733')  # Hexadecimal color code
plt.title('Hexadecimal Color Example')
plt.show()
```

- Hexadecimal codes allow for precise control over the color (e.g., '#FF5733' for a specific shade of red-orange).

---

3. Customizing Line Style and Markers

```
plt.plot(x, y, color='green', linestyle='--', marker='o')  # Dashed line with circle markers
plt.title('Line Style and Markers')
plt.show()
```

- linestyle: Can be 'solid', 'dashed', 'dotted', or 'dashdot'.
- marker: Choose from symbols like 'o' (circle), 'x' (cross), 's' (square), etc.

---

4. Multiple Lines with Different Styles

```
y2 = [1, 2, 3, 4]
plt.plot(x, y, color='blue', linestyle='-', marker='o', label='Line 1')
plt.plot(x, y2, color='red', linestyle='--', marker='s', label='Line 2')

plt.title('Multiple Lines with Different Styles')
plt.legend()  # Show legend
plt.show()
```

- You can plot multiple lines with different styles and use plt.legend() to label each line.

---

5. Customizing Background Color

```
fig, ax = plt.subplots()
ax.plot(x, y, color='blue')
fig.patch.set_facecolor('#f2f2f2')  # Set background color of the entire figure

ax.set_title('Custom Background Color')
plt.show()
```

- Use fig.patch.set_facecolor() to change the background color of the figure or axes.

---

6. Using Colormaps for Continuous Data

```
import numpy as np

z = np.linspace(0, 10, 100)
y3 = np.sin(z)

plt.scatter(z, y3, c=z, cmap='viridis')  # Use a colormap for continuous data
plt.colorbar()  # Show colorbar
plt.title('Scatter Plot with Colormap')
plt.show()
```

- cmap='viridis': Applies a colormap to the scatter plot.
- Matplotlib has many predefined colormaps like 'viridis', 'plasma', and 'inferno'.

Key Takeaways:

- **Colors**: Use named colors, RGB tuples, or hexadecimal color codes to customize your plot.
- **Line Style and Markers**: Change the appearance of your plot using linestyle and marker.

- **Colormaps**: For continuous data, use colormaps like viridis to represent values visually.
- **Background Color**: Customize the plot background color to match your desired style.

Interview Questions (Medium to Hard):

Q1: How would you change the color of a line in Matplotlib? What are some ways to specify the color?
A:

- You can specify the color using the color parameter in plt.plot(). Colors can be specified as named colors (e.g., 'blue'), RGB tuples (e.g., (0.1, 0.2, 0.5)), or hexadecimal color codes (e.g., '#FF5733').

Q2: How would you make the plot background color different from the default white?
A:

- You can change the figure background color using fig.patch.set_facecolor(). For example, fig.patch.set_facecolor('#f2f2f2') changes the background color to a light gray.

Q3: You have two lines to plot on the same graph, and you want one line to be dashed and the other solid. How would you accomplish this?
A:

- You can use the linestyle parameter in plt.plot(). For example, use linestyle='-' for the solid line and linestyle='--' for the dashed line.

Q4: What is the purpose of cmap in Matplotlib, and how do you use it in scatter plots?
A:

- cmap is used to apply a colormap to continuous data, allowing you to visually differentiate values. You can use it with scatter plots to color points based on their value, e.g., plt.scatter(x, y, c=z, cmap='viridis') where z is the data to be mapped to colors.

Q5: How would you display a color legend on a scatter plot when using a colormap?
A:

- You can use plt.colorbar() to display the color legend or scale for the colormap in a scatter plot.

✅ **End of Day 8!**

Day 9: Subplots and Multiple Axes in Matplotlib

Why Use Subplots?

- **Comparison**: Plot multiple datasets side by side to compare trends or patterns.
- **Space Management**: Keep your plots organized without cluttering one figure.
- **Flexibility**: Customize each subplot independently (different scales, labels, etc.).

---

1. Creating Subplots with plt.subplot()

```
import matplotlib.pyplot as plt

# Create a 2x2 grid of subplots
plt.subplot(2, 2, 1)  # Row 1, Column 1
plt.plot([1, 2, 3], [1, 4, 9], 'r')

plt.subplot(2, 2, 2)  # Row 1, Column 2
plt.plot([1, 2, 3], [1, 2, 3], 'g')

plt.subplot(2, 2, 3)  # Row 2, Column 1
plt.plot([1, 2, 3], [9, 4, 1], 'b')

plt.subplot(2, 2, 4)  # Row 2, Column 2
plt.plot([1, 2, 3], [1, 3, 2], 'k')

plt.show()
```

- plt.subplot(nrows, ncols, index) divides the figure into nrows rows and ncols columns, and index specifies the subplot position (1-based index).

---

2. Subplots with plt.subplots()

The plt.subplots() method is a more flexible way to create subplots, returning a figure and an array of axes.

```
fig, axs = plt.subplots(2, 2)  # 2x2 grid
axs[0, 0].plot([1, 2, 3], [1, 4, 9], 'r')
axs[0, 1].plot([1, 2, 3], [1, 2, 3], 'g')
axs[1, 0].plot([1, 2, 3], [9, 4, 1], 'b')
axs[1, 1].plot([1, 2, 3], [1, 3, 2], 'k')

plt.tight_layout()  # Adjust layout to prevent overlapping labels
plt.show()
```

- This approach returns fig (the figure object) and axs (an array of axes), which you can use to plot on specific subplots.

---

3. Customizing Individual Subplots

```
fig, axs = plt.subplots(2, 1)  # Two subplots, stacked vertically

axs[0].plot([1, 2, 3], [1, 4, 9], 'r')
axs[0].set_title('Plot 1')
axs[0].set_xlabel('X-axis')
axs[0].set_ylabel('Y-axis')

axs[1].plot([1, 2, 3], [1, 2, 3], 'g')
axs[1].set_title('Plot 2')
axs[1].set_xlabel('X-axis')
axs[1].set_ylabel('Y-axis')

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

- You can customize each subplot individually, changing titles, labels, etc.

---

4. Sharing Axes Between Subplots

```
fig, axs = plt.subplots(2, 1, sharex=True, sharey=True)  # Share x and y axes

axs[0].plot([1, 2, 3], [1, 4, 9], 'r')
axs[1].plot([1, 2, 3], [1, 2, 3], 'g')

axs[0].set_title('Shared Axes Plot')
plt.tight_layout()
plt.show()
```

- sharex=True and sharey=True allow you to link the x-axis or y-axis between subplots, making comparisons easier.

---

5. Specifying Size of Subplots with figsize

```
fig, axs = plt.subplots(2, 2, figsize=(10, 6))  # Adjust figure size
axs[0, 0].plot([1, 2, 3], [1, 4, 9], 'r')
axs[0, 1].plot([1, 2, 3], [1, 2, 3], 'g')
axs[1, 0].plot([1, 2, 3], [9, 4, 1], 'b')
axs[1, 1].plot([1, 2, 3], [1, 3, 2], 'k')

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

- figsize=(width, height) adjusts the overall size of the figure.

Key Takeaways:

- Use plt.subplot() for basic subplot creation with a single function call.
- plt.subplots() returns a figure and axes, allowing for more customization.
- You can share axes between subplots to make comparisons easier.
- Use figsize to control the size of the overall figure.

Interview Questions (Medium to Hard):

Q1: When would you use plt.subplot() vs. plt.subplots()?
A:

- plt.subplot() is great for quick, simple subplots when you want to divide the figure into rows and columns with a specific index. It's useful for basic layouts.
- plt.subplots() is more flexible, returning both the figure and axes, allowing you to customize each subplot and manage layouts with ease. It's ideal for complex or multiple subplots.

Q2: How would you make two plots in a grid with shared x-axis?
A:

- Use plt.subplots() with sharex=True. For example: fig, axs = plt.subplots(2, 1, sharex=True) will create two subplots stacked vertically with a shared x-axis.

Q3: How do you prevent overlap in subplot labels and titles?
A:

- You can use plt.tight_layout() to automatically adjust subplot parameters (e.g., labels, titles) to fit within the figure area without overlap.

Q4: How would you adjust the size of subplots to fit more details?
A:

- You can use the figsize parameter in plt.subplots(), e.g., fig, axs = plt.subplots(2, 2, figsize=(10, 6)), to control the figure size for better visualization.

Q5: How can you share both x and y axes between multiple subplots?
A:

- When creating subplots with plt.subplots(), you can use sharex=True and sharey=True to share both axes between subplots. This is useful for comparing trends across multiple subplots with the same axis scales.

✅ **End of Day 9!**

Day 10: Advanced Plotting Techniques in Matplotlib

Why Advanced Plotting?

- **3D Plots**: Visualize data in three dimensions to represent more complex datasets.
- **Annotations**: Add labels, arrows, and other markers to make your plot more informative.
- **Customizing Legends**: Improve the clarity of your plots by customizing how legends are displayed.

---

1. 3D Plotting with Axes3D

To plot 3D data, you need to import the Axes3D module and create a 3D axes object.

```
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
import numpy as np

# Create data
x = np.linspace(-5, 5, 100)
y = np.linspace(-5, 5, 100)
x, y = np.meshgrid(x, y)
z = np.sin(np.sqrt(x**2 + y**2))

# Create figure and 3D axes
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

# Plot 3D surface
ax.plot_surface(x, y, z, cmap='viridis')

plt.show()
```

- **Axes3D** is used to create 3D axes.
- **plot_surface()** allows you to plot a 3D surface, and cmap controls the color map.

---

2. Adding Annotations

Annotations help highlight important points or areas in your plot.

```
x = [1, 2, 3, 4]
y = [10, 20, 25, 30]

plt.plot(x, y, 'bo-')
```

```
# Add annotation
plt.annotate('Max Point', xy=(3, 25), xytext=(2, 28),
             arrowprops=dict(facecolor='black', arrowstyle='->'),
             fontsize=12)

plt.show()
```

- **annotate()** adds text annotations. You can also add an arrow with arrowprops to point to a specific location.

---

3. Customizing Legends

Legends are crucial for understanding what each plot represents. Customizing legends can make them more informative and visually appealing.

```
# Create data
x = np.linspace(0, 10, 100)
y1 = np.sin(x)
y2 = np.cos(x)

# Plot with legend
plt.plot(x, y1, label='Sine', color='r')
plt.plot(x, y2, label='Cosine', color='b')

# Customize legend
plt.legend(loc='upper right', fontsize=12, title='Trigonometric Functions')

plt.show()
```

- **plt.legend()** adds the legend, where loc specifies the location and title adds a title to the legend.
- You can also change the font size and style with fontsize.

---

4. Customizing Axis Ticks and Labels

You can modify the tick marks and labels on both x and y axes for better clarity.

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

plt.plot(x, y)

# Customize x and y ticks
plt.xticks([0, 2, 4, 6, 8, 10], ['0', '2', '4', '6', '8', '10'])
plt.yticks([-1, 0, 1], ['Low', 'Zero', 'High'])

plt.show()
```

- **xticks() and yticks()** allow you to change the tick positions and labels.

5. Plotting Multiple Lines with Different Styles

You can plot multiple datasets on the same figure with different styles, colors, and markers.

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

plt.plot(x, y1, label='Sine Wave', linestyle='-', color='r', marker='o')
plt.plot(x, y2, label='Cosine Wave', linestyle='--', color='b', marker='x')

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

- Customize line style, color, and marker type using linestyle, color, and marker parameters.

Key Takeaways:

- Use **3D plotting** to visualize data in three dimensions with Axes3D.
- **Annotations** help highlight key points or areas with text and arrows.
- **Customizing legends** improves plot clarity and makes it easier to understand different datasets.
- **Axis customization** can make your plots more readable and tailored to your needs.

Interview Questions (Medium to Hard):

Q1: How would you create a 3D surface plot in Matplotlib?
A:

- Use from mpl_toolkits.mplot3d import Axes3D to enable 3D plotting.
- Create a 3D axes object with fig.add_subplot(111, projection='3d').
- Plot the surface using ax.plot_surface(x, y, z) where x, y, and z are the coordinates.

Q2: What's the purpose of annotations, and how can you add an arrow to highlight a point?
A:

- Annotations are used to add textual information to specific points in the plot, improving its clarity.
- Use plt.annotate() with the arrowprops parameter to add arrows pointing to specific data points.

Q3: How can you improve the appearance of legends in a plot?
A:

- Customize the legend using plt.legend(), where you can adjust the location with loc, add a title with title, and set font size with fontsize. You can also control legend markers and colors.

Q4: How do you adjust the ticks and labels on both x and y axes in Matplotlib?
A:

- Use plt.xticks() and plt.yticks() to modify tick positions and labels. You can specify both the tick positions and their corresponding labels.

Q5: Can you explain how to plot multiple datasets with different line styles and markers?
A:

- Use the plt.plot() function for each dataset, and customize the line style with linestyle, the color with color, and markers with marker. You can then add a legend to differentiate between the datasets.

✅ **End of Day 10!**

Day 11: Working with Date and Time in Matplotlib

Why Plot Time Series Data?

- **Trend Analysis**: Time series allows you to observe data trends over time (e.g., stock prices, temperature changes).
- **Data Intervals**: You can easily plot data points that correspond to specific time intervals.
- **Annotations and Markers**: Highlight key events or changes over time.

---

1. Plotting Time Series Data

For time series data, the **x-axis** typically represents time, while the **y-axis** represents the value of the variable you are tracking.

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

# Create time series data
date_range = pd.date_range(start='2022-01-01', periods=10, freq='D')
data = [10, 15, 20, 25, 18, 22, 24, 30, 35, 40]

# Create a DataFrame
df = pd.DataFrame({'Date': date_range, 'Value': data})

# Plot the data
plt.plot(df['Date'], df['Value'], marker='o')
plt.title('Time Series Plot')
plt.xlabel('Date')
plt.ylabel('Value')
plt.xticks(rotation=45)  # Rotate x-axis labels for better readability
plt.tight_layout()  # Adjust layout to avoid clipping
```

```
plt.show()
```

- **pd.date_range()** creates a range of dates.
- **plt.plot()** is used for plotting, with dates on the x-axis.
- **xticks(rotation=45)** rotates date labels to improve visibility.

---

2. Customizing Date Format on the x-axis

You can format the date axis using mdates.DateFormatter for better readability.

```
import matplotlib.dates as mdates

# Plot the data again
plt.plot(df['Date'], df['Value'], marker='o')

# Customize the date format on the x-axis
plt.gca().xaxis.set_major_formatter(mdates.DateFormatter('%Y-%m-%d'))

# Rotate x-axis labels for better readability
plt.xticks(rotation=45)

plt.show()
```

- **mdates.DateFormatter()** allows you to specify a custom format for the dates (e.g., year-month-day).
- You can modify the format to display more or less information, such as just the year ('%Y').

---

3. Plotting with Resampling

Often, time series data is in higher frequency (e.g., minute-wise) and you might want to resample it to a lower frequency (e.g., daily averages).

```
# Create data with higher frequency
date_range = pd.date_range(start='2022-01-01', periods=100, freq='H')
data = range(100)

df = pd.DataFrame({'Date': date_range, 'Value': data})

# Resample to daily frequency and calculate mean
df_resampled = df.resample('D', on='Date').mean()

# Plot the resampled data
plt.plot(df_resampled.index, df_resampled['Value'], marker='o')
plt.title('Resampled Time Series (Daily Average)')
plt.xlabel('Date')
plt.ylabel('Value')
plt.xticks(rotation=45)
plt.tight_layout()
plt.show()
```

- **resample()** is used to change the frequency of the time series. 'D' stands for daily frequency.
- The **mean** is calculated for each day in the resampled data.

---

4. Highlighting Specific Date Ranges

Sometimes, you may want to highlight specific periods in your time series plot (e.g., anomalies or special events).

```
# Highlight a specific date range
plt.plot(df['Date'], df['Value'], marker='o')

# Highlight the range between two dates with a shaded region
highlight = [(3, 6)]  # Indexes of the date range
for start, end in highlight:
    plt.axvspan(df['Date'][start], df['Date'][end], color='yellow', alpha=0.3)

plt.title('Highlighted Time Series Data')
plt.xlabel('Date')
plt.ylabel('Value')
plt.xticks(rotation=45)
plt.tight_layout()
plt.show()
```

- **axvspan()** creates a vertical span (highlight) on the plot.
- You can adjust **alpha** for transparency to blend the highlighted area with the rest of the plot.

---

5. Multiple Time Series on the Same Plot

It’s common to compare multiple time series on the same plot.

```
# Create two time series
data2 = [30, 35, 40, 45, 50, 55, 60, 65, 70, 75]
df2 = pd.DataFrame({'Date': date_range, 'Value': data2})

# Plot both time series
plt.plot(df['Date'], df['Value'], label='Series 1', color='r', marker='o')
plt.plot(df2['Date'], df2['Value'], label='Series 2', color='b', marker='x')

# Add legend
plt.legend()

plt.title('Multiple Time Series')
plt.xlabel('Date')
plt.ylabel('Value')
plt.xticks(rotation=45)
plt.tight_layout()
plt.show()
```

- **plt.legend()** is used to distinguish between the two time series.

Key Takeaways:

- Use **pd.date_range()** to create time series data, and plot it using plt.plot().
- Customize the **x-axis** for date formatting using mdates.DateFormatter().
- **Resampling** allows you to change the frequency of your time series data (e.g., from hourly to daily).
- You can **highlight specific date ranges** with axvspan() to focus on important periods.
- **Multiple time series** can be compared in the same plot, helping visualize relationships between different datasets.

Interview Questions (Medium to Hard):

Q1: How do you format the date on the x-axis in a Matplotlib plot?
A:

- Use matplotlib.dates.DateFormatter to format the date axis, specifying the desired format (e.g., '%Y-%m-%d').

Q2: How would you handle a high-frequency time series and resample it to a lower frequency?
A:

- Use df.resample('D', on='Date').mean() to resample the time series to a daily frequency, calculating the mean for each day.

Q3: How can you highlight a specific date range in a time series plot?
A:

- Use plt.axvspan(start_date, end_date, color='color', alpha=0.3) to highlight a date range by adding a shaded region.

Q4: Can you explain how to plot multiple time series on the same plot?
A:

- You can plot multiple time series by calling plt.plot() for each series and use plt.legend() to distinguish between them.

Q5: What is the purpose of resampling in time series analysis, and how is it done in Pandas?
A:

- Resampling changes the frequency of time series data (e.g., from hourly to daily). It’s done using the resample() function, which allows you to aggregate or interpolate the data.

✅ **End of Day 11!**

Day 12: Creating Subplots for Multi-Plot Layouts in Matplotlib

Why Use Subplots?

- **Compact Visualization**: Display multiple charts in a single figure to save space and make comparisons.
- **Ease of Comparison**: Place related plots next to each other for side-by-side comparison.
- **Organized Presentation**: Make your visualizations cleaner and more organized.

---

1. Basic Subplot Layout

The **plt.subplot()** function is used to create subplots. It takes 3 arguments: the number of rows, the number of columns, and the index of the subplot.

```
import matplotlib.pyplot as plt

# Create a 1x2 grid of subplots
plt.subplot(1, 2, 1)  # First subplot
plt.plot([1, 2, 3], [4, 5, 6])
plt.title('Plot 1')

plt.subplot(1, 2, 2)  # Second subplot
plt.plot([1, 2, 3], [6, 5, 4])
plt.title('Plot 2')

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

- **plt.subplot(1, 2, 1)** creates a subplot grid of 1 row and 2 columns, and the 1 refers to the position of the first plot.
- **plt.subplot(1, 2, 2)** creates the second plot in the grid.

---

2. Advanced Subplot Layout with plt.subplots()

plt.subplots() is a more flexible and convenient method to create a grid of subplots. It returns a **figure object** and an array of **axes** objects, which can be used to control the individual plots.

```
# Create a 2x2 grid of subplots
fig, axes = plt.subplots(2, 2)

# Plot on the first subplot
axes[0, 0].plot([1, 2, 3], [4, 5, 6])
axes[0, 0].set_title('Plot 1')

# Plot on the second subplot
axes[0, 1].plot([1, 2, 3], [6, 5, 4])
```

```
axes[0, 1].set_title('Plot 2')

# Plot on the third subplot
axes[1, 0].plot([1, 2, 3], [7, 8, 9])
axes[1, 0].set_title('Plot 3')

# Plot on the fourth subplot
axes[1, 1].plot([1, 2, 3], [9, 8, 7])
axes[1, 1].set_title('Plot 4')

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

- **fig, axes = plt.subplots(2, 2)** creates a 2x2 grid of subplots, where axes is a 2D array of subplot axes.
- Each subplot is accessed via the **axes array** (e.g., axes[0, 0] refers to the first subplot).

---

3. Sharing Axes Across Subplots

Sometimes, you might want to share the x-axis or y-axis across multiple subplots for easier comparison. This can be done using the **sharex** and **sharey** parameters in **plt.subplots()**.

```
# Create a 2x2 grid of subplots, sharing the x-axis
fig, axes = plt.subplots(2, 2, sharex=True)

axes[0, 0].plot([1, 2, 3], [4, 5, 6])
axes[0, 0].set_title('Plot 1')

axes[0, 1].plot([1, 2, 3], [6, 5, 4])
axes[0, 1].set_title('Plot 2')

axes[1, 0].plot([1, 2, 3], [7, 8, 9])
axes[1, 0].set_title('Plot 3')

axes[1, 1].plot([1, 2, 3], [9, 8, 7])
axes[1, 1].set_title('Plot 4')

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

- **sharex=True** ensures all subplots share the same x-axis, making it easier to compare data across subplots.
- Similarly, **sharey=True** can be used to share the y-axis.

---

4. Customizing Subplots with gridspec

gridspec gives you more control over subplot positioning and sizes.

```
import matplotlib.gridspec as gridspec

# Create a figure and a gridspec layout
fig = plt.figure(figsize=(8, 6))
gs = gridspec.GridSpec(2, 2, figure=fig)

# Create subplots using the grid
ax1 = fig.add_subplot(gs[0, 0])
ax1.plot([1, 2, 3], [4, 5, 6])
ax1.set_title('Plot 1')

ax2 = fig.add_subplot(gs[0, 1])
ax2.plot([1, 2, 3], [6, 5, 4])
ax2.set_title('Plot 2')

ax3 = fig.add_subplot(gs[1, :])  # Span across the entire second row
ax3.plot([1, 2, 3], [7, 8, 9])
ax3.set_title('Plot 3')

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

- **gridspec.GridSpec()** creates a flexible grid layout. You can span subplots across multiple rows or columns using slicing, such as gs[1, :] (spanning the second row).

---

5. Plotting with Different Figure Sizes

You can set a custom size for the entire figure using the figsize argument in **plt.subplots()** or **fig.figure(figsize=(width, height))**.

```
fig, axes = plt.subplots(2, 2, figsize=(10, 8))  # Customize figure size
axes[0, 0].plot([1, 2, 3], [4, 5, 6])
axes[0, 0].set_title('Plot 1')

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

- **figsize=(width, height)** is used to control the size of the entire figure.

Key Takeaways:

- **plt.subplot()** is used for basic subplot layouts with specific grid sizes.
- **plt.subplots()** is more flexible and returns both a figure and axes objects for easy subplot manipulation.
- You can **share axes** between subplots to align data for easier comparison.

- **gridspec** offers advanced customization of subplot placement and sizing.
- Adjusting the **figure size** can help in presenting complex multi-plot layouts.

Interview Questions (Medium to Hard):

Q1: How does plt.subplots() differ from plt.subplot() in terms of functionality and flexibility?
A:

- plt.subplots() is more flexible, returning both the figure and axes objects, allowing easier customization of multiple subplots. It also supports shared axes.
- plt.subplot() creates individual subplots but requires manual handling of subplot positions, and it's less flexible for complex layouts.

Q2: What is the advantage of using sharex=True or sharey=True in subplots?
A:

- These parameters ensure that the x-axis or y-axis is shared across all subplots, making it easier to compare data visually without axis mismatches. It also saves space and keeps the plot layout cleaner.

Q3: How would you use gridspec to customize the layout of subplots in a figure?
A:

- gridspec.GridSpec() allows creating custom grid layouts where subplots can span multiple rows or columns. You can create non-uniform layouts, such as having one subplot span two columns or a row.

Q4: What would you do if you need a larger space for one of your subplots in a multi-plot layout?
A:

- You can adjust the size of subplots using **gridspec** or by manually setting the position of that subplot using **plt.subplots_adjust()** or by adjusting **figsize**.

Q5: How do you adjust the spacing between subplots to avoid overlapping plots or labels?
A:

- Use **plt.tight_layout()** to automatically adjust spacing between subplots, ensuring that they do not overlap. You can also manually adjust spacing using **plt.subplots_adjust()**.

✅ **End of Day 12!**

Day 13: Customizing Plot Styles and Themes in Matplotlib

Why Customize Plot Styles and Themes?

- **Visual Appeal**: Make your plots more aesthetically pleasing for better communication.
- **Consistency**: Use a consistent style across all visualizations in a project or report.
- **Highlight Key Insights**: Custom styles can help emphasize specific parts of the plot, such as trends or outliers.

---

1. Using Built-in Styles

Matplotlib comes with a variety of pre-defined styles that can be applied to your plots using **plt.style.use()**.

Example of Built-in Styles:

```
import matplotlib.pyplot as plt

# Set the 'ggplot' style (a popular style inspired by R's ggplot2)
plt.style.use('ggplot')

# Plot with ggplot style
plt.plot([1, 2, 3], [4, 5, 6])
plt.title('Plot with ggplot Style')
plt.show()
```

Matplotlib includes several styles such as:

- 'seaborn-darkgrid'
- 'bmh'
- 'ggplot'
- 'classic'
- 'fivethirtyeight'

To see a list of available styles:

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

---

2. Customizing Plot Elements

You can modify individual elements of your plot, such as the color, line style, marker style, and more, by using the corresponding parameters.

Example of Customizing Line and Marker Styles:

```
import matplotlib.pyplot as plt

# Customize line style, color, and marker
plt.plot([1, 2, 3], [4, 5, 6], linestyle='--', color='green', marker='o')
```

```
plt.title('Custom Line and Marker Style')
plt.show()
```

- **linestyle**: Choose from '-', '--', ':', etc.
- **color**: Set the color of the line (e.g., 'blue', 'red', '#FF5733').
- **marker**: Choose a marker (e.g., 'o' for circles, 's' for squares).

---

3. Customizing Axes and Gridlines

You can modify axes limits, labels, and gridlines to improve the readability of the plot.

Example of Customizing Axes and Gridlines:

```
import matplotlib.pyplot as plt

# Create a plot
plt.plot([1, 2, 3], [4, 5, 6])

# Customize x and y axes labels
plt.xlabel('X-Axis Label')
plt.ylabel('Y-Axis Label')

# Customize the axes limits
plt.xlim(0, 4)
plt.ylim(0, 7)

# Add gridlines with specific style
plt.grid(True, linestyle=':', color='purple')

plt.title('Customized Axes and Gridlines')
plt.show()
```

- **xlabel()** and **ylabel()**: Set the x and y axis labels.
- **xlim()** and **ylim()**: Set custom axis limits.
- **plt.grid()**: Customize gridlines (e.g., linestyle, color).

---

4. Changing Font Properties

You can customize font properties such as size, family, and weight for titles, labels, and ticks.

Example of Customizing Font Properties:

```
import matplotlib.pyplot as plt

# Create a plot
plt.plot([1, 2, 3], [4, 5, 6])

# Set title and axis labels with custom fonts
plt.title('Custom Font Properties', fontsize=16, family='serif', fontweight='bold')
```

```
plt.xlabel('X-Axis', fontsize=12, family='sans-serif')
plt.ylabel('Y-Axis', fontsize=12, family='sans-serif')

plt.show()
```

- **fontsize**: Controls the size of the text.
- **family**: Sets the font family (e.g., 'serif', 'sans-serif').
- **fontweight**: Sets the font weight (e.g., 'bold').

---

5. Creating and Using Custom Styles

You can create your own custom styles by creating a .mplstyle file, which contains settings for various elements of the plot.

Example of Creating a Custom Style File:

1. Create a new file named my_style.mplstyle with the following content:

```
axes.titlesize: 18
axes.labelsize: 14
axes.labelweight: bold
axes.grid: True
grid.linestyle: '-'
grid.color: blue
lines.linewidth: 2
lines.color: green
```

2. Use your custom style in your script:

```
import matplotlib.pyplot as plt

# Apply custom style from 'my_style.mplstyle' file
plt.style.use('my_style.mplstyle')

plt.plot([1, 2, 3], [4, 5, 6])
plt.title('Plot with Custom Style')
plt.show()
```

By creating your own style, you can reuse it across different plots for consistency.

---

6. Saving Plots with Custom Styles

After customizing your plot, you can save it to a file with a specific format and resolution using **plt.savefig()**.

Example of Saving a Plot:

```
import matplotlib.pyplot as plt

# Create a plot
plt.plot([1, 2, 3], [4, 5, 6])
```

```
# Save the plot as a PNG image with 300 dpi
plt.savefig('custom_plot.png', dpi=300)

plt.show()
```

- **dpi**: Controls the resolution of the saved plot.
- **transparent**: Can be set to True to save with a transparent background.

Key Takeaways:

- **Use built-in styles** with **plt.style.use()** to quickly change the look of your plots.
- **Customize individual plot elements** like lines, markers, axes, and gridlines for more control over the visual appearance.
- **Create custom styles** by defining your own .mplstyle file for consistent formatting across plots.
- **Customize font properties** to adjust text size, family, and weight.
- **Save plots** with **plt.savefig()** to export them in high resolution and various formats.

🧠 **Interview Questions (Medium to Hard):**

Q1: How can you create and apply your own custom style in Matplotlib?
A:

- Create a .mplstyle file that contains custom settings for plot elements like axes, gridlines, lines, and fonts. Use plt.style.use('my_style.mplstyle') to apply the style to a plot.

Q2: When would you use **plt.tight_layout()** in relation to customizing plot styles, and why?
A:

- **plt.tight_layout()** is used to adjust the spacing between subplots or plot elements (like titles, labels, and ticks). It ensures that nothing overlaps and the plot looks neat, especially after customizing the layout and adding multiple elements.

Q3: How would you change the font style of axis labels and title in a Matplotlib plot?
A:

- Use **fontsize**, **family**, and **fontweight** parameters in functions like plt.title(), plt.xlabel(), and plt.ylabel() to customize the font size, family, and weight of titles and labels.

Q4: What is the purpose of **sharex** and **sharey** when creating multiple subplots, and how would you use them?
A:

- **sharex=True** and **sharey=True** are used to ensure that all subplots share the same x-axis or y-axis, respectively. This is particularly useful when comparing data across subplots as it avoids inconsistent axis scaling.

Q5: How would you save a plot with a transparent background and high resolution?
A:

- Use **plt.savefig('filename.png', dpi=300, transparent=True)** to save the plot with 300 dpi resolution and a transparent background.

✅ **End of Day 13!**

Day 14: Advanced Plotting Techniques in Matplotlib

Why Advanced Plotting Techniques?

- **Complex Data Visualization**: Handle large datasets and complex relationships.
- **Interactivity**: Add interactivity to your plots for dynamic data exploration.
- **Customization**: Customize every aspect of your plot to suit the context of the analysis.

---

1. Creating Subplots

Matplotlib allows you to create multiple plots within a single figure using **plt.subplots()**. This is especially useful for comparing multiple datasets side by side.

Example of Creating Subplots:

```
import matplotlib.pyplot as plt

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

# Plot on each subplot
axs[0, 0].plot([1, 2, 3], [4, 5, 6])
axs[0, 0].set_title('Plot 1')

axs[0, 1].bar([1, 2, 3], [4, 5, 6])
axs[0, 1].set_title('Plot 2')

axs[1, 0].scatter([1, 2, 3], [4, 5, 6])
axs[1, 0].set_title('Plot 3')

axs[1, 1].hist([1, 2, 2, 3, 3, 3], bins=3)
axs[1, 1].set_title('Plot 4')

plt.tight_layout()  # Adjust layout for better spacing
plt.show()
```

- **fig, axs = plt.subplots()**: Creates a figure and a grid of subplots.
- **axs[row, col].plot()**: Use the specific subplot axis to plot the graph.

---

2. Polar Plots

Polar plots are useful for data that has a cyclic or angular nature, such as angles, directions, and periodic data.

Example of Creating a Polar Plot:

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

# Data for the polar plot
theta = np.linspace(0, 2*np.pi, 100)
r = np.sin(theta)  # Example of periodic data

# Create a polar plot
plt.subplot(projection='polar')
plt.plot(theta, r)

plt.title('Polar Plot')
plt.show()
```

- **plt.subplot(projection='polar')**: Defines the polar plot.
- **r = np.sin(theta)**: Example of data on the polar coordinate system.

---

3. 3D Plots

Matplotlib allows you to create 3D plots for visualizing data in three dimensions. You can use this for 3D scatter plots, surface plots, and more.

Example of Creating a 3D Scatter Plot:

```
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import numpy as np

# Data for the 3D scatter plot
x = np.random.rand(100)
y = np.random.rand(100)
z = np.random.rand(100)

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

# Scatter plot
ax.scatter(x, y, z)
```

```
ax.set_title('3D Scatter Plot')
plt.show()
```

- **from mpl_toolkits.mplot3d import Axes3D**: Import the 3D plotting module.
- **ax.scatter(x, y, z)**: Create a 3D scatter plot using x, y, and z data.

---

4. Annotating Plots

You can add annotations to highlight specific points or areas of interest in your plots.

Example of Annotating a Plot:

```
import matplotlib.pyplot as plt

# Plot data
x = [1, 2, 3, 4, 5]
y = [2, 3, 5, 7, 11]
plt.plot(x, y)

# Annotate a specific point
plt.annotate('Prime number', xy=(3, 5), xytext=(4, 6),
             arrowprops=dict(facecolor='red', arrowstyle='->'))

plt.title('Annotated Plot')
plt.show()
```

- **plt.annotate()**: Adds an annotation to a plot. You can specify the text, position, and an arrow to point to a specific data point.

---

5. Adding Interactive Widgets (Matplotlib + Jupyter)

Matplotlib can be made interactive using widgets in Jupyter notebooks. This allows users to change plot parameters interactively, such as adjusting axis limits or plot types.

Example of Creating an Interactive Plot:

```
import matplotlib.pyplot as plt
from ipywidgets import interact
import numpy as np

# Function to update the plot based on input values
def update_plot(freq=1):
    t = np.linspace(0, 2*np.pi, 100)
    plt.plot(t, np.sin(freq * t))
    plt.title(f'Sine Wave with Frequency {freq}')
    plt.show()

# Create an interactive widget
interact(update_plot, freq=(1, 10, 0.1))
```

- **ipywidgets.interact()**: Creates an interactive widget to update the plot based on user input.
- **freq**: Adjusts the frequency of the sine wave interactively.

---

6. Heatmaps

Heatmaps are a great way to visualize matrix-like data or 2D datasets, showing how values vary across the matrix with colors.

Example of Creating a Heatmap:

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

# Create random data
data = np.random.rand(10, 10)

# Create a heatmap
plt.imshow(data, cmap='viridis', interpolation='nearest')
plt.colorbar()  # Show color scale
plt.title('Heatmap')
plt.show()
```

- **plt.imshow()**: Displays an image (in this case, a heatmap) from 2D data.
- **cmap='viridis'**: Defines the color map (you can choose from different options like coolwarm, plasma, etc.).

Key Takeaways:

- **Subplots**: Use **plt.subplots()** to create multiple plots in a single figure for comparison.
- **Polar Plots**: Use **plt.subplot(projection='polar')** to create plots based on polar coordinates.
- **3D Plots**: Use **Axes3D** for creating 3D scatter, surface, and line plots.
- **Annotations**: Use **plt.annotate()** to add text and arrows to highlight points in a plot.
- **Interactivity**: Use **ipywidgets** to make plots interactive in Jupyter notebooks.
- **Heatmaps**: Use **plt.imshow()** to visualize 2D data with colors representing values.

🧠 **Interview Questions (Medium to Hard):**

Q1: How would you create a 3D surface plot in Matplotlib?
A:

- Import Axes3D from mpl_toolkits.mplot3d. Create a 3D subplot using fig.add_subplot(111, projection='3d'). Use ax.plot_surface() with x, y, and z data to create the surface plot.

Q2: Can you explain how **plt.subplots()** works, and what parameters are used to control the layout of subplots?
A:

- **plt.subplots()** creates a figure and an array of subplots. You can control the layout using parameters like nrows, ncols, and sharex/sharey to share axes between subplots.

Q3: What is the difference between **imshow()** and **scatter()** in Matplotlib?
A:

- **imshow()** is used to display image-like data (2D arrays), while **scatter()** is used to create scatter plots with individual data points (usually in 2D or 3D).

Q4: How would you make your plot interactive in Jupyter Notebooks?
A:

- Use **ipywidgets** to create interactive widgets (like sliders, buttons) that allow you to dynamically update the plot based on user input.

Q5: What are the common color maps in heatmaps, and how would you choose one?
A:

- Common color maps include **viridis**, **plasma**, **inferno**, **coolwarm**, etc. The choice depends on the nature of the data, e.g., coolwarm for data with both negative and positive values, and viridis for visually pleasant continuous data.

✅ **End of Day 14!**

Day 15: Customizing Legends and Annotations in Matplotlib

Why Customize Legends and Annotations?

- **Clarity**: Make plots easier to understand by clearly labeling the elements.
- **Information**: Provide additional context or explanations directly on the plot.
- **Aesthetics**: Enhance the visual appeal and readability of plots.

---

1. Customizing Legends

Legends help explain what each plot element (line, scatter point, bar, etc.) represents. Matplotlib allows you to customize the position, title, fonts, and more for legends.

Basic Legend:

import matplotlib.pyplot as plt

```
x = [1, 2, 3, 4]
y1 = [1, 4, 9, 16]
y2 = [1, 2, 3, 4]

plt.plot(x, y1, label='Quadratic')
plt.plot(x, y2, label='Linear')

# Add legend
plt.legend()

plt.show()
```

Customizing Legend Position and Style:

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

# Customizing legend
plt.legend(loc='upper left', fontsize='large', title='Function Types')

plt.show()
```

- **loc**: Defines the position of the legend (e.g., 'upper left', 'lower right').
- **fontsize**: Customizes the font size.
- **title**: Adds a title to the legend.

---

2. Legends with Multiple Elements

If you have multiple elements in a plot, you can create more sophisticated legends with groupings or selective entries.

Example of Multiple Legends:

```
import matplotlib.pyplot as plt

x = [1, 2, 3, 4]
y1 = [1, 4, 9, 16]
y2 = [1, 2, 3, 4]
y3 = [1, 3, 6, 10]

plt.plot(x, y1, label='Quadratic')
plt.plot(x, y2, label='Linear')
plt.plot(x, y3, label='Cubic')

# Create a legend with selective items
plt.legend(handles=[plt.Line2D(x, y1, label='Quadratic'), plt.Line2D(x, y3, label='Cubic')])

plt.show()
```

- **handles**: Manually specify which plot elements should appear in the legend.

---

3. Customizing Annotations

Annotations can be used to add text or highlight specific points on the plot, which can help emphasize key insights.

Example of Simple Annotation:

```
import matplotlib.pyplot as plt

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

plt.plot(x, y)

# Annotating a point
plt.annotate('Highest point', xy=(4, 16), xytext=(3, 15),
             arrowprops=dict(facecolor='red', arrowstyle='->'))

plt.show()
```

- **xy**: Specifies the point to annotate.
- **xytext**: Specifies where the annotation text should appear.
- **arrowprops**: Customizes the appearance of the arrow (e.g., color, style).

---

4. Advanced Annotations with Multiple Texts

You can also annotate multiple points or add text in different positions to provide further context.

Example of Multiple Annotations:

```
import matplotlib.pyplot as plt

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

plt.plot(x, y)

# Annotate multiple points
plt.annotate('Start point', xy=(1, 1), xytext=(2, 2),
             arrowprops=dict(facecolor='blue', arrowstyle='->'))
plt.annotate('End point', xy=(4, 16), xytext=(3, 14),
             arrowprops=dict(facecolor='green', arrowstyle='->'))

plt.show()
```

- You can add multiple annotations with different **xy** and **xytext** positions.

---

5. Annotating with Mathematical Equations

Matplotlib allows you to use LaTeX formatting in annotations to add mathematical expressions.

Example of Annotating with LaTeX:

```
import matplotlib.pyplot as plt

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

plt.plot(x, y)

# Annotate with LaTeX equation
plt.annotate(r'$y = x^2$', xy=(2, 4), xytext=(3, 10), fontsize=12,
             arrowprops=dict(facecolor='red', arrowstyle='->'))

plt.show()
```

- **r'$y = x^2$'**: The r before the string allows you to use raw string literals, and the LaTeX expression is placed between dollar signs ($).

---

6. Customizing Text Placement

Matplotlib provides various options for controlling text placement on the plot.

Example of Customizing Text:

```
import matplotlib.pyplot as plt

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

plt.plot(x, y)

# Place text on the plot with specific alignment
plt.text(2, 10, 'Peak Point', fontsize=14, ha='center', va='top')

plt.show()
```

- **ha**: Horizontal alignment ('left', 'center', 'right').
- **va**: Vertical alignment ('top', 'center', 'bottom').

Key Takeaways:

- **Legends**: Customizing legends improves plot readability. Use plt.legend() to control position, fonts, and labels.
- **Annotations**: Use **plt.annotate()** to add context to specific points or regions on the plot.
- **Multiple Elements**: Legends and annotations can handle multiple items for better clarity.
- **LaTeX in Annotations**: You can use LaTeX to display mathematical formulas in annotations.
- **Text Placement**: Customize text placement with alignment options (ha, va).

Interview Questions (Medium to Hard):

Q1: How can you place multiple legends in a plot, each referring to different parts of the plot?
A:

- You can use **plt.legend()** with the **handles** parameter, which allows you to select specific lines or elements to include in the legend. Alternatively, you can create multiple legends using **plt.legend()** for different regions of the plot.

Q2: What is the difference between using **annotate()** and **text()** for adding labels in Matplotlib?
A:

- **annotate()** is used when you want to point to a specific data point with an arrow or other visual cues, whereas **text()** is used for placing arbitrary text at a specific location on the plot without pointing to any data point.

Q3: How would you display a mathematical equation in the annotation?
A:

- Use LaTeX formatting by placing the equation inside dollar signs (e.g., r'$y = x^2$') when passing the string to **plt.annotate()**.

Q4: What is the importance of customizing legend positions, and how can you do it?
A:

- Customizing the position of legends ensures they do not overlap with the data. Use the **loc** parameter in **plt.legend()** to control the legend's position, e.g., 'upper left', 'center right'.

Q5: How do you highlight a point in your plot using annotation, and what parameters do you need to control for customizing the arrow?

A:

- Use **plt.annotate()** with the xy parameter for the point, xytext for the text position, and arrowprops to customize the arrow's appearance (e.g., color, style, width).

✅ **End of Day 15!**

📅 **Day 16: Stacked Bar Charts**

Why Use Stacked Bar Charts?

- **Stacked bar charts** are ideal when you want to display the composition of different categories and show how subgroups contribute to a total.
- They are useful for comparing parts of a whole across different categories.

---

1. Creating Stacked Bar Charts

To create stacked bar charts in Matplotlib, you can use the **stacked=True** parameter in the **bar()** function.

Example: Basic Stacked Bar Chart

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

# Data
categories = ['A', 'B', 'C']
group1 = [3, 4, 2]
group2 = [1, 2, 3]

# Plotting the stacked bar chart
plt.bar(categories, group1, label='Group 1')
plt.bar(categories, group2, bottom=group1, label='Group 2')

plt.title('Stacked Bar Chart')
plt.xlabel('Category')
plt.ylabel('Values')
plt.legend()

plt.show()
```

In this example, **Group 1** is plotted first, and **Group 2** is stacked on top of it using the bottom parameter.

---

2. Customizing Stacked Bar Charts

You can further customize stacked bar charts by adjusting the colors and labels.
Example: Custom Stacked Bar Chart

```
plt.bar(categories, group1, label='Group 1', color='skyblue')
plt.bar(categories, group2, bottom=group1, label='Group 2', color='orange')

plt.title('Custom Stacked Bar Chart')
plt.xlabel('Category')
plt.ylabel('Values')
plt.legend()

plt.show()
```

Key Takeaways:

- **Stacked Bar Charts** are useful for comparing the contribution of different subgroups to a total.
- Use **bottom** parameter to stack the bars on top of each other.
- Customize the chart with **colors** and **labels** for better readability.

Interview Questions (Medium to Hard):

Q1: When would you choose a stacked bar chart over a regular bar chart?
A:

- Use stacked bar charts when you need to show the total value along with the composition of different categories. Regular bar charts are better when you want to compare absolute values between categories.

Q2: How can you handle negative values in a stacked bar chart?
A:

- You can use the **bottom** parameter to adjust the positioning of negative bars, ensuring they are stacked below the x-axis.

Day 17: Subplots and Multiple Axes

Why Use Multiple Subplots?

- Multiple subplots allow you to visualize different aspects of your data in one figure.
- **Subplots** are great when comparing multiple plots side by side or showing different views of the same data.

---

1. Creating Multiple Subplots

Use **plt.subplots()** to create a grid of subplots.
Example: Basic Subplots

```
fig, ax = plt.subplots(1, 2)  # 1 row, 2 columns

ax[0].plot([1, 2, 3], [1, 4, 9], label='Plot 1')
ax[1].bar(['A', 'B', 'C'], [3, 4, 5], label='Plot 2')

ax[0].set_title('Line Plot')
ax[1].set_title('Bar Plot')

plt.show()
```

This creates a figure with two subplots arranged in a single row (1x2 grid).

---

2. Adjusting the Layout of Subplots

You can adjust the spacing and layout using **fig.tight_layout()**.

Example: Adjusting Layout

```
fig, ax = plt.subplots(1, 2)
ax[0].plot([1, 2, 3], [1, 4, 9])
ax[1].bar(['A', 'B', 'C'], [3, 4, 5])

fig.tight_layout()  # Adjust layout to avoid overlap

plt.show()
```

Key Takeaways:

- **Subplots** allow multiple plots to share the same figure.
- Use **tight_layout()** to adjust spacing between subplots.
- Customize each subplot independently while sharing a common figure.

Interview Questions (Medium to Hard):

Q1: How would you arrange a grid of 3x3 subplots in Matplotlib?
A:

- You would use **plt.subplots(3, 3)** to create a 3x3 grid of subplots.

Q2: How do you share x and y axes between subplots?
A:

- Use the **sharex** and **sharey** parameters in **plt.subplots()** to share axes between subplots.

Day 18: Scatter Plots

Why Use Scatter Plots?

- Scatter plots are ideal for visualizing the relationship between two continuous variables.
- They help detect patterns, trends, and outliers.

---

1. Creating Scatter Plots

Use **scatter()** to create scatter plots.

Example: Basic Scatter Plot

```
x = [1, 2, 3, 4, 5]
y = [1, 4, 9, 16, 25]

plt.scatter(x, y)
plt.title('Basic Scatter Plot')
plt.xlabel('X')
plt.ylabel('Y')
plt.show()
```

---

2. Customizing Scatter Plots

You can customize the marker style, size, and color.

Example: Custom Scatter Plot

```
plt.scatter(x, y, color='red', marker='x', s=100)
plt.title('Customized Scatter Plot')
plt.xlabel('X')
plt.ylabel('Y')
plt.show()
```

- **color**: Sets the color of the points.
- **marker**: Defines the shape of the scatter points.
- **s**: Adjusts the size of the markers.

Key Takeaways:

- **Scatter plots** are great for visualizing the correlation between two continuous variables.
- You can customize the markers, size, and colors for better data presentation.

Interview Questions (Medium to Hard):

Q1: How would you visualize the relationship between three continuous variables?
A:

- Use a **3D scatter plot** (Axes3D), where each axis represents a different variable.

Q2: How do you add a regression line to a scatter plot?
A:

- You can use **seaborn.regplot()** or fit a regression line manually and plot it using **plot()**.

Day 19: Pie Charts

Why Use Pie Charts?

- **Pie charts** are ideal for visualizing parts of a whole in categorical data.
- They show the relative proportion of each category as a slice of the pie.

---

1. Creating Pie Charts

Use **pie()** to create a pie chart.

Example: Basic Pie Chart

```
labels = ['A', 'B', 'C', 'D']
sizes = [15, 30, 45, 10]

plt.pie(sizes, labels=labels, autopct='%1.1f%%')
plt.title('Basic Pie Chart')
plt.show()
```

- **autopct**: Adds percentage labels to each slice.
- **labels**: Specifies the categories for each slice.

---

2. Customizing Pie Charts

You can adjust the colors, explode slices, and start the angle of the chart.

Example: Customized Pie Chart

```
explode = (0.1, 0, 0, 0)  # 'explode' the first slice
colors = ['gold', 'yellowgreen', 'lightcoral', 'lightskyblue']

plt.pie(sizes, labels=labels, autopct='%1.1f%%', explode=explode, colors=colors)
plt.title('Customized Pie Chart')
plt.show()
```

Key Takeaways:

- **Pie charts** are best for showing the proportional distribution of categories.
- Customize slice colors, labels, and the angle of the chart to improve visualization.

Interview Questions (Medium to Hard):

Q1: When would you avoid using a pie chart?
A:

- Avoid pie charts if there are too many categories (more than 5-6), as it becomes hard to compare slices effectively.

Q2: How can you handle the situation where one category has a very small percentage?
A:

- Consider using a **bar chart** instead or combine small categories into an "Other" category.

Day 20: Creating Line Plots

Why Use Line Plots?

- **Line plots** are excellent for visualizing trends over time or continuous data.
- They are simple and effective for showing the relationship between two variables.

---

1. Creating Basic Line Plots
Use **plot()** to create line plots.

Example: Basic Line Plot

```
x = [1, 2, 3, 4, 5]
y = [1, 4, 9, 16, 25]

plt.plot(x, y)
plt.title('Basic Line Plot')
plt.xlabel('X')
plt.ylabel('Y')
plt.show()
```

---

2. Customizing Line Plots

You can customize line style, color, and markers.

Example: Customized Line Plot

```
plt.plot(x, y, linestyle='--', color='r', marker='o')
plt.title('Customized Line Plot')
plt.xlabel('X')
```

```
plt.ylabel('Y')
plt.show()
```

Key Takeaways:

- **Line plots** are best for visualizing trends in continuous data.
- Customize lines with **linestyle**, **color**, and **marker** for better clarity.

🧠 **Interview Questions (Medium to Hard):**

Q1: When should you use a line plot instead of a bar chart?
A:

- **Line plots** are for continuous data (e.g., time-series), while **bar charts** are used for categorical data comparison.

Q2: How can you display multiple line plots in one graph?
A:

- Simply call **plot()** multiple times for each dataset.