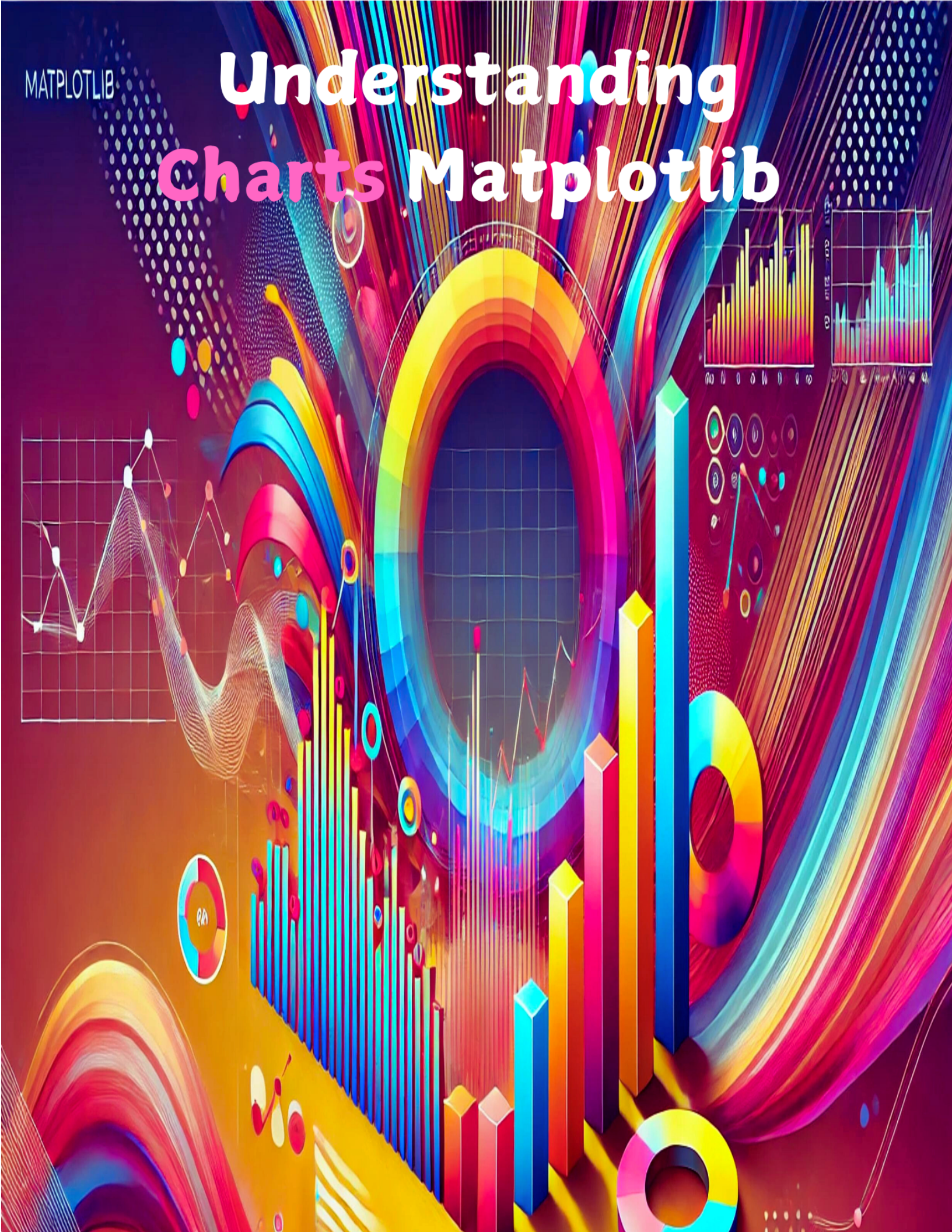

MATPLOTLIB
Understanding
Charts Matplotlib

Let's start with the **Line Plot**, and I'll guide you through a simple example first, followed by more advanced customization to make it visually appealing.

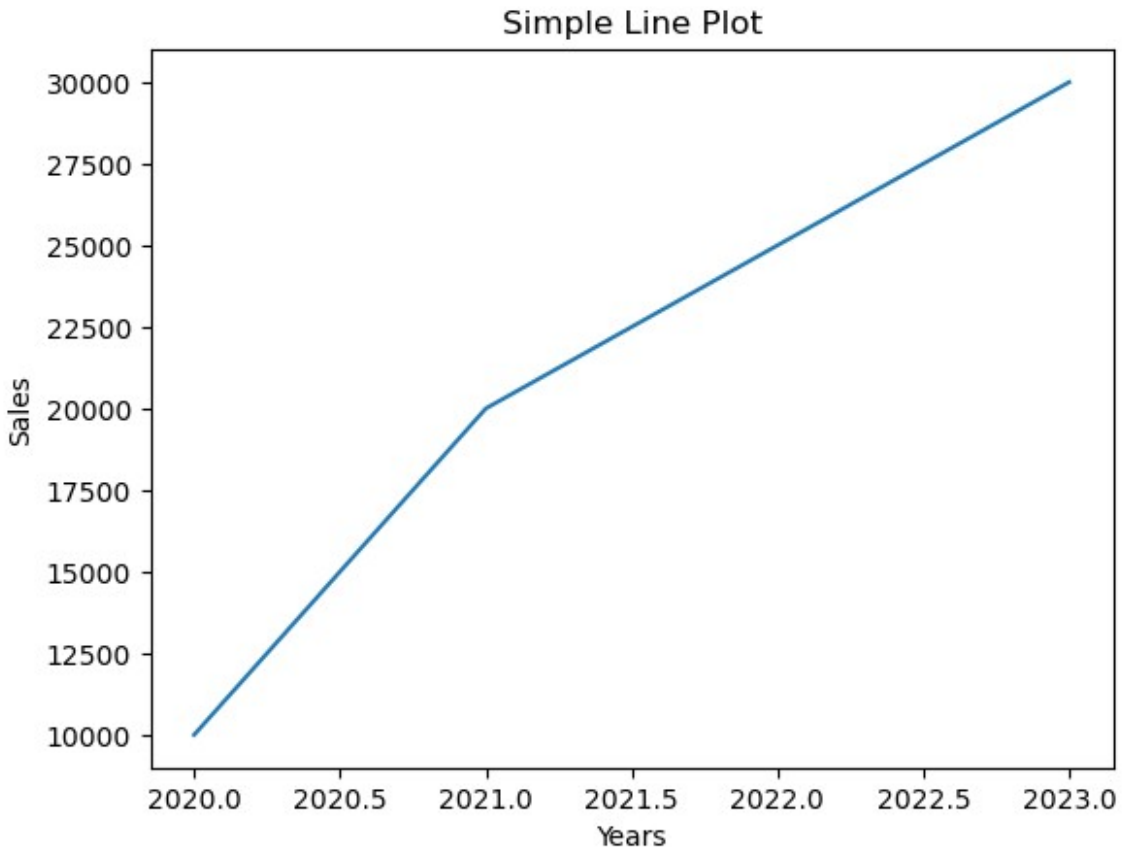
1. Line Plot

A **line plot** is used to show data points connected by straight lines. It's great for showing trends over time or relationships between two variables.

Step 1: Simple Line Plot Example

Here's how to create a basic line plot using `matplotlib.pyplot`:

```
import matplotlib.pyplot as plt

# Data for the plot
x = [2020,2021,2022,2023]  # X-axis data (e.g., years)
y = [10000, 20000, 25000, 30000]  # Y-axis data (e.g., sales)

# Create the line plot
plt.plot(x, y)

# Add labels and a title
plt.xlabel('Years')
plt.ylabel('Sales')
plt.title('Simple Line Plot')

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



Explanation:

- `x = [1, 2, 3, 4, 5]` represents the years (or some sequence) on the X-axis.
- `y = [10, 20, 25, 30, 35]` represents the sales (or some values) on the Y-axis.
- `plt.plot(x, y)`: Creates a line plot with the given `x` and `y` values.
- `plt.xlabel()` and `plt.ylabel()` set the labels for the X and Y axes.
- `plt.title()` gives the plot a title.
- `plt.show()` displays the plot.

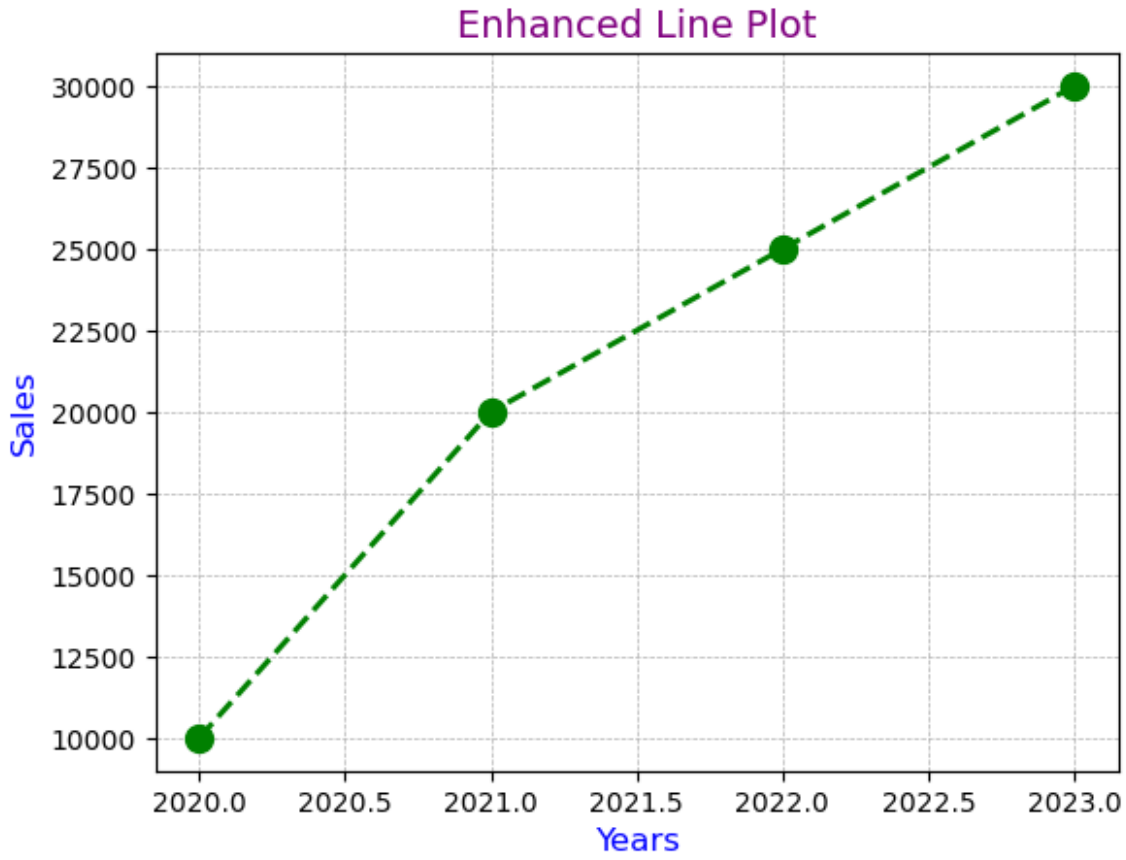
Output: This will produce a simple line plot with the x-values on the horizontal axis and the y-values on the vertical axis.

Step 2: Enhancing the Line Plot

Now, let's make the plot more visually appealing by adding colors, markers, line styles, and a grid.

```
import matplotlib.pyplot as plt

# Data for the plot
x = [2020,2021,2022,2023]  # X-axis data (e.g., years)
y = [10000, 20000, 25000, 30000]  # Y-axis data (e.g., sales)
```

```
# Create the line plot with enhancements
plt.plot(x, y, color='green', linestyle='--', marker='o',
markersize='10',linewidth=2)

# Add labels and a title
plt.xlabel('Years', fontsize=12, color='blue')
plt.ylabel('Sales', fontsize=12, color='blue')
plt.title('Enhanced Line Plot', fontsize=14, color='purple')

# Add grid and set line properties
plt.grid(True, which='both', linestyle='--', linewidth=0.5)

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



Enhancements Explained:

- **`color='green'`**: Changes the color of the line to green.
- **`linestyle='--'`**: Changes the line style to dashed (`--`), but you can also use other styles like `'-'` (solid) or `':'` (dotted).
- **`marker='o'`**: Adds circular markers at each data point.
- **`markersize=10`**: Increases the thickness of the marker.
- **`linewidth=2`**: Increases the thickness of the line.

- **`fontsize=12`**: Increases the font size for labels.
- **`color='blue'/color='purple'`**: Changes the color of the axis labels and title.
- **`plt.grid()`**: Adds a grid to the plot, making it easier to read the data. The `linestyle='--'` and `linewidth=0.5` customize the grid style.

Output: This enhanced plot will look more professional, with a green dashed line, markers on data points, and a grid. The axis labels and title are colored and styled to stand out.

Why Use Line Plots?

- **Trend Analysis:** Line plots are great for showing how a variable changes over time or comparing the relationships between two numerical variables.
- **Customization:** You can easily customize line plots to improve clarity and presentation.

Common Customization Options:

- **Color:** You can change the line color by specifying `color='color_name'`.
- **Line Style:** Modify the style of the line (`'-'` for solid, `'--'` for dashed, `':'` for dotted).
- **Markers:** Add markers to indicate data points (`'o'` for circles, `'s'` for squares, etc.).
- **Grid:** Add a grid using `plt.grid()`, which makes it easier to interpret the data.

Next Steps:

- **Do you feel comfortable with the line plot?** If yes, let me know, and we can move on to the next type of plot, such as a bar chart.
- **Any questions about the line plot?** Feel free to ask for clarification on any part!

Sure! Here's a list of common enhancements you can apply to a **line plot** in `matplotlib` to make it more visually appealing and informative:

Common Enhancements for Line Charts:

1. **Line Style (`linestyle`)**: Change the style of the line (`'-'` for solid, `'--'` for dashed, `':'` for dotted, etc.).
2. **Line Width (`linewidth`)**: Control the thickness of the line.
3. **Line Color (`color`)**: Set the color of the line.
4. **Markers (`marker`)**: Add markers at each data point (`'o'` for circle, `'s'` for square, etc.).
5. **Marker Size (`markersize`)**: Adjust the size of the markers.
6. **Marker Color (`markerfacecolor`, `markeredgecolor`)**: Customize the color of the marker's face and edge.
7. **Marker Edge Width (`markeredgewidth`)**: Set the thickness of the marker's edge.
8. **Grid (`plt.grid()`)**: Add a grid to the plot.
9. **Axis Labels (`xlabel`, `ylabel`)**: Set the labels for the X and Y axes.
10. **Title (`title`)**: Set a title for the plot.
11. **Legend (`plt.legend()`)**: Add a legend for the lines if there are multiple lines plotted.

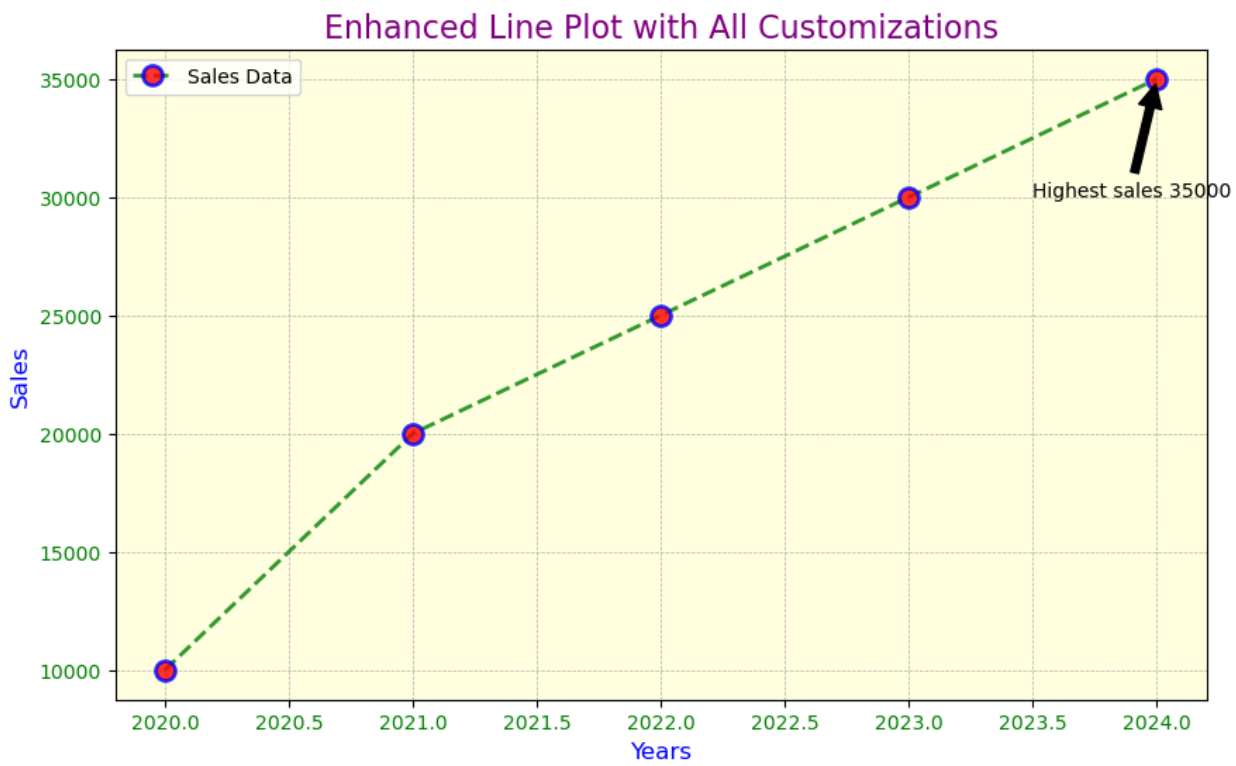
12. **Ticks Customization (`xticks`, `yticks`)**: Customize the tick marks and their labels on both axes.
13. **Figure Size (`plt.figure(figsize=(width, height))`)**: Set the overall size of the plot.
14. **Axis Limits (`plt.xlim()`, `plt.ylim()`)**: Control the limits of the X and Y axes.
15. **Alpha (`alpha`)**: Set the transparency level of the line (0 for transparent, 1 for opaque).
16. **Line Style Patterns (`dash_capstyle`, `dash_joinstyle`)**: Customize how dashed lines are rendered at joins and ends.
17. **Annotations (`plt.annotate()`)**: Add text annotations at specific data points.
18. **Background Color (`plt.gca().set_facecolor()`)**: Set the background color of the plot area.
19. **Line Labels (`label`)**: Assign labels to lines for the legend.
20. **Subplots (`plt.subplot()`)**: Add multiple plots in a single figure.

Code Example with All Enhancements:

Here's a comprehensive example that uses most of these enhancements:

```
# Data for the plot
x = [2020,2021,2022,2023,2024]  # X-axis data (e.g., years)
y = [10000, 20000, 25000, 30000,35000]  # Y-axis data (e.g., sales)

# Set the figure size
plt.figure(figsize=(10, 6))

# Create the enhanced line plot
plt.plot(x, y,
         color='green',              # Line color
         linestyle='--',             # Line style: dashed
         marker='o',                 # Marker type: circle
         markersize=10,              # Marker size
         markerfacecolor='red',      # Marker face color
         markeredgecolor='blue',     # Marker edge color
         markeredgewidth=2,          # Marker edge width
         linewidth=2,                # Line width
         alpha=0.8,                  # Line transparency
         label='Sales Data')         # Line label for legend

# Add labels and a title
plt.xlabel('Years', fontsize=12, color='blue')
plt.ylabel('Sales', fontsize=12, color='blue')
plt.title('Enhanced Line Plot with All Customizations', fontsize=16,
color='purple')

# Add grid with customization
plt.grid(True, which='both', linestyle='--', linewidth=0.5)

# Customize the ticks on both axes
```

```
plt.xticks(color='green', fontsize=10)
# plt.xticks(ticks=x, labels=['Year 1', 'Year 2', 'Year 3', 'Year 4',
'Year 5'], fontsize=10)
plt.yticks(color='green', fontsize=10)
# plt.yticks(ticks=range(5000, 40000, 50000), fontsize=10)

# # Set axis limits
# plt.xlim(2019, 2025)
# plt.ylim(0, 40000)

# Add a legend
plt.legend(loc='upper left', fontsize=10)

# Add an annotation at a specific point
plt.annotate('Highest sales 35000', xy=(2024, 35000), xytext=(2023.5,
30000),
             arrowprops=dict(facecolor='black', shrink=0.05))

# Change the background color of the plot area
plt.gca().set_facecolor('lightyellow')

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



Explanation of Enhancements:

1. **Line Style and Color:**

- `color='green'`: Sets the line color to green.
- `linestyle='--'`: Changes the line to a dashed style.
- `linewidth=2`: Makes the line thicker.

2. **Markers:**
   - `marker='o'`: Adds circular markers to each data point.
   - `markersize=10`: Increases the size of the markers.
   - `markerfacecolor='red'`: Fills the markers with red.
   - `markeredgecolor='blue'`: Colors the edges of the markers blue.
   - `markeredgewidth=2`: Thickens the edge of the markers.
3. **Grid and Axis Customization:**
   - `plt.grid()`: Adds a grid with dashed lines and thin width.
   - `plt.xticks()` and `plt.yticks()`: Customize the tick marks on both the X and Y axes.
4. **Legend and Labels:**
   - `label='Sales Data'`: Assigns a label to the line, which appears in the legend.
   - `plt.legend()`: Displays the legend in the upper left corner.
5. **Annotations:**
   - `plt.annotate()`: Adds a text annotation at a specific point (x=4, y=30) with an arrow pointing to it.
6. **Plot Area Customization:**
   - `plt.gca().set_facecolor('lightyellow')`: Changes the background color of the plot to light yellow.
7. **Title and Axis Labels:**
   - `plt.title()` and `plt.xlabel()/plt.ylabel()`: Customize the font size and color of the title and axis labels.
8. **Transparency:**
   - `alpha=0.8`: Adds transparency to the line, making it 80% opaque.

Summary of Enhancements:

Enhancement	Description
Line Style	Changes the style of the line (`'--'`, `'-'`, etc.).
Line Width	Adjusts the thickness of the line.
Color	Sets the line color.
Markers	Adds markers to highlight data points.
Marker Size/Color	Adjusts the size and color of the markers.
Grid	Adds a grid to the plot.
Labels	Adds titles, axis labels, and a legend.
Ticks	Customizes the tick marks on the X and Y axes.
Figure Size	Adjusts the overall plot size.
Annotations	Adds annotations to mark specific points.
Background Color	Changes the background color of the plot area.

Enhancement	Description
Alpha (Transparency)	Adds transparency to the line.

- You can try running this code and modify specific parts to see how the plot changes.

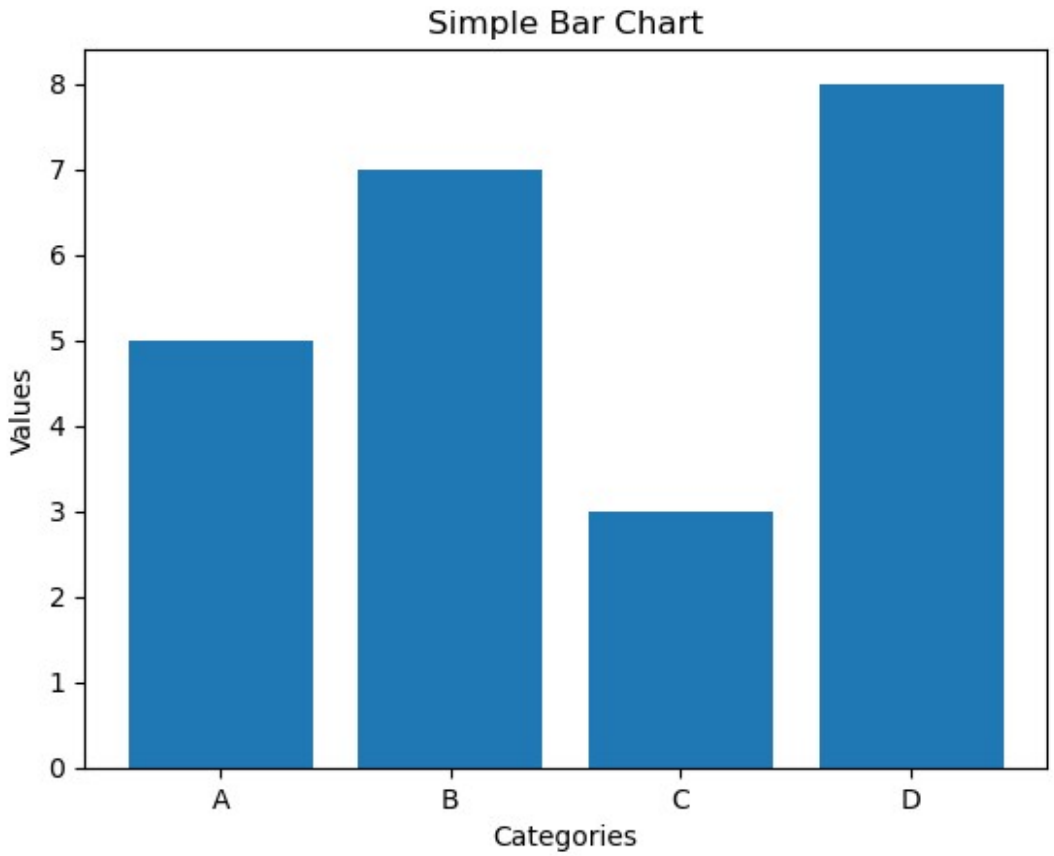
2. Bar Chart

A **bar chart** is used to represent categorical data with rectangular bars. The length of each bar is proportional to the value it represents. Bar charts are excellent for comparing values across categories.

Step 1: Simple Bar Chart Example

Here's how to create a basic bar chart using `matplotlib.pyplot`:

```
import matplotlib.pyplot as plt

# Data for the plot
categories = ['A', 'B', 'C', 'D']
values = [5, 7, 3, 8]

# Create the bar chart
plt.bar(categories, values)

# Add labels and a title
plt.xlabel('Categories')
plt.ylabel('Values')
plt.title('Simple Bar Chart')

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



Explanation:

- `categories = ['A', 'B', 'C', 'D']`: These are the categories, or labels, for each bar.
- `values = [5, 7, 3, 8]`: These are the values corresponding to each category.
- `plt.bar(categories, values)`: Creates a bar chart with the specified categories on the x-axis and their corresponding values as the height of the bars.
- `plt.show()`: Displays the plot.

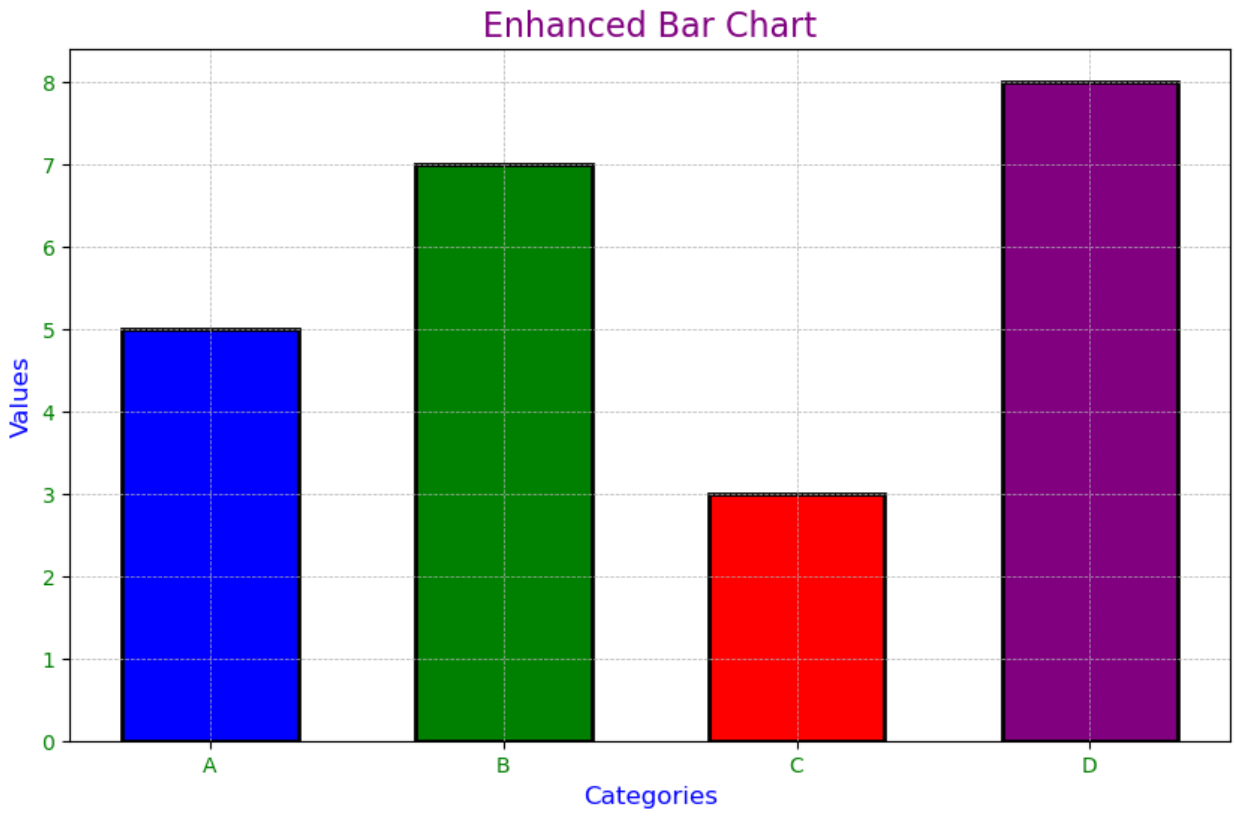
Output: This will produce a simple bar chart with categories A, B, C, and D on the x-axis and the values on the y-axis.

Step 2: Enhancing the Bar Chart

Now, let's make the bar chart more visually appealing by adding colors, changing the width of the bars, adding a grid, and customizing labels.

```
import matplotlib.pyplot as plt

# Data for the plot
categories = ['A', 'B', 'C', 'D']
values = [5, 7, 3, 8]
```

```
# Set the figure size
plt.figure(figsize=(10, 6))

# Create an enhanced bar chart with customizations
bars = plt.bar(categories, values,
        color=['blue', 'green', 'red', 'purple'],  # Set colors for
each bar
        edgecolor='black',                         # Set the color of
the bar edges
        linewidth=2,                               # Thickness of the
edges
        width=0.6)                                 # Width of the bars

# Create a line plot on top of the bar chart
# plt.plot(categories, values,
#          color='red',
#          marker='o',
#          linestyle='-',
#          linewidth=2,
#          markersize=8,
#          label='Line Data')  # Add a label for the line chart

# Show values on top of the bars
# for bar in bars:
#     yval = bar.get_height()  # Get the height of the bar
#     plt.text(bar.get_x() + bar.get_width()/2, yval,  # Set the
position of the text
#              f'{yval}',  # Text to display (the value of the bar)
#              ha='center', va='bottom',  # Center the text
horizontally and place it just above the bar
#              fontsize=10, color='black')  # Customize the appearance
of the text

# Add labels and a title with custom font sizes and colors
plt.xlabel('Categories', fontsize=12, color='blue')
plt.ylabel('Values', fontsize=12, color='blue')
plt.title('Enhanced Bar Chart', fontsize=16, color='purple')

# Add gridlines
plt.grid(True, which='both', linestyle='--', linewidth=0.5)

# Customize the ticks
plt.xticks(fontsize=10, color='green')
plt.yticks(fontsize=10, color='green')

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



Enhancements Explained:

- `color=['blue', 'green', 'red', 'purple']`: Sets a different color for each bar.
- `edgecolor='black'`: Adds a black outline around each bar.
- `linewidth=2`: Increases the thickness of the edges around the bars.
- `width=0.6`: Reduces the width of each bar (default is 0.8).
- `fontsize=12`/`fontsize=16`: Customizes the font size of the labels and title.
- `plt.grid()`: Adds a grid with dashed lines for better readability.
- `plt.xticks()`/`plt.yticks()`: Customizes the font size and color of the tick marks on the X and Y axes.

Output: This enhanced bar chart will have custom bar colors, a black outline, a grid, and well-labeled axes.

Why Use Bar Charts?

- **Category Comparison:** Bar charts are great for comparing the values of different categories side by side.
- **Customization:** You can easily customize bar charts to improve readability and presentation, such as changing colors, adding gridlines, and adjusting bar widths.

Common Customization Options for Bar Charts:

1. **Bar Colors (`color`)**: Change the color of the bars.
2. **Edge Color (`edgecolor`)**: Set the color of the bar borders.
3. **Bar Width (`width`)**: Adjust the width of the bars (default is 0.8).
4. **Orientation (`plt.barh()`)**: Create horizontal bar charts instead of vertical ones.
5. **Gridlines (`plt.grid()`)**: Add gridlines for better readability.
6. **Labels and Title (`xlabel`, `ylabel`, `title`)**: Customize the labels and title with font sizes and colors.
7. **Custom Ticks (`xticks`, `yticks`)**: Customize the tick marks on the X and Y axes.
8. **Adding Values to Bars**: Annotate bars with their actual values.
9. **Transparency (`alpha`)**: Add transparency to the bars.
10. **Figure Size (`plt.figure(figsize=(width, height))`)**: Adjust the size of the plot.

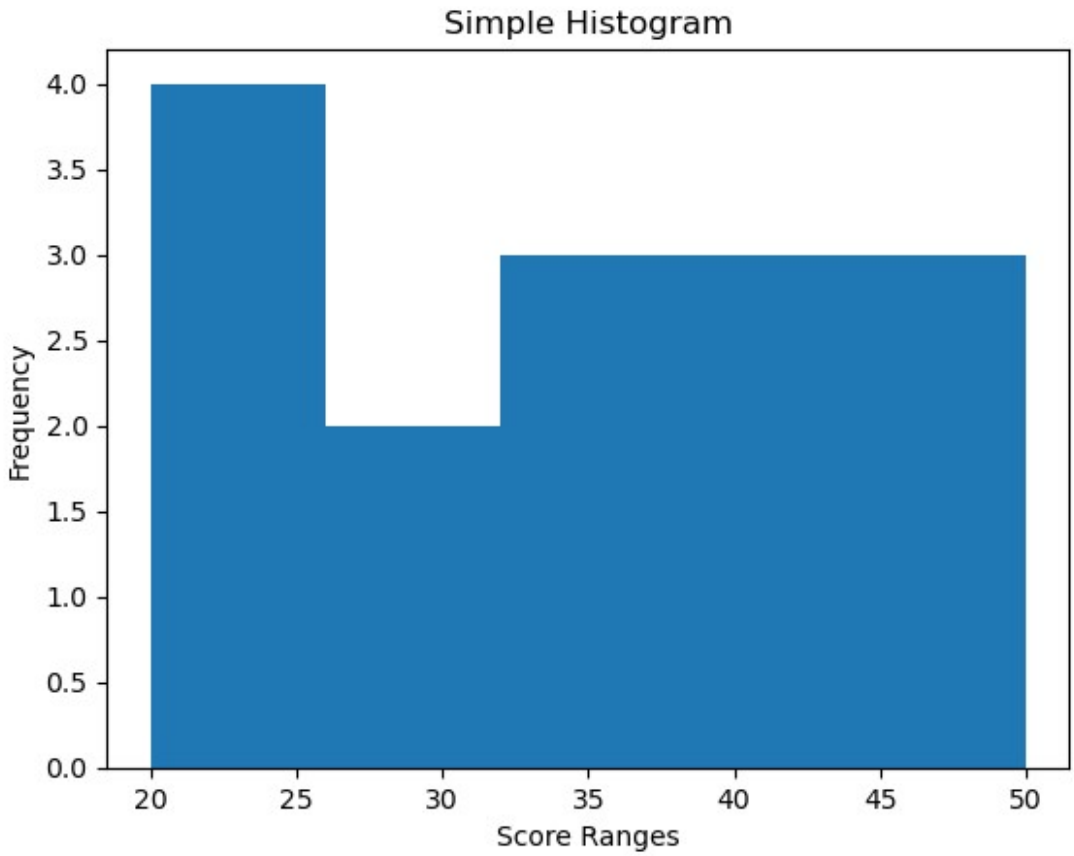
3. Histogram

A **histogram** is used to represent the distribution of a dataset. It groups data into **bins** (intervals) and counts how many data points fall into each bin. Histograms are useful for understanding the frequency distribution of numerical data.

Step 1: Simple Histogram Example

Here's how to create a basic histogram using `matplotlib.pyplot`:

```
import matplotlib.pyplot as plt

# Data for the histogram (e.g., test scores)
data = [20, 23, 25, 25, 28, 30, 32, 35, 35, 38, 40, 42, 45, 48, 50]

# Create a simple histogram
plt.hist(data, bins=5)

# Add labels and a title
plt.xlabel('Score Ranges')
plt.ylabel('Frequency')
plt.title('Simple Histogram')

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



Explanation:

- **`data`**: This is the dataset you want to visualize (e.g., test scores).
- **`plt.hist(data, bins=5)`**: Creates a histogram with 5 bins (ranges). Each bin shows how many data points fall within that range.
- **`plt.show()`**: Displays the plot.

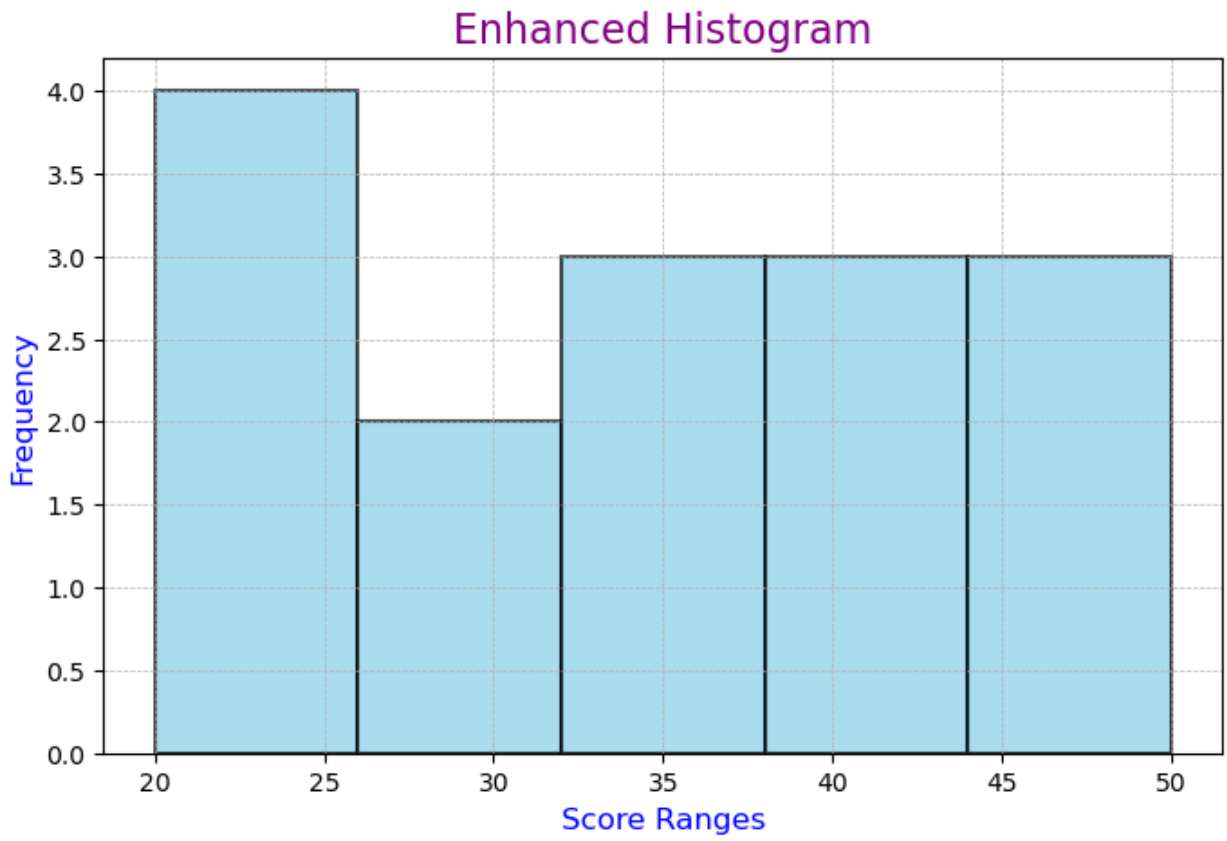
Output: You will see a histogram where the x-axis shows the score ranges, and the y-axis shows the number of data points (frequency) in each range.

Step 2: Enhancing the Histogram

Let's make the histogram more visually appealing by adding custom colors, edge lines, transparency, and gridlines.

```
import matplotlib.pyplot as plt

# Data for the histogram (e.g., test scores)
data = [20, 23, 25, 25, 28, 30, 32, 35, 35, 38, 40, 42, 45, 48, 50]

# Set the figure size
plt.figure(figsize=(8, 5))
```

```
# Create an enhanced histogram with customizations
plt.hist(data,
         bins=5,                        # Set the number of bins
         color='skyblue',               # Set bar color
         edgecolor='black',             # Set edge color for the bars
         alpha=0.7,                     # Set transparency
         linewidth=1.5)                 # Thickness of the edges

# Add labels and a title
plt.xlabel('Score Ranges', fontsize=12, color='blue')
plt.ylabel('Frequency', fontsize=12, color='blue')
plt.title('Enhanced Histogram', fontsize=16, color='purple')

# Add gridlines
plt.grid(True, which='both', linestyle='--', linewidth=0.5)

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



Enhancements Explained:

- **`bins=5`**: Specifies that the data should be grouped into 5 bins (you can adjust this based on how granular you want the ranges).

- **`color='skyblue'`**: Changes the color of the bars to sky blue.
- **`edgecolor='black'`**: Adds black edges around each bin for better visibility.
- **`alpha=0.7`**: Adds transparency to the bars, allowing you to see overlapping data better.
- **`linewidth=1.5`**: Increases the thickness of the bar edges.
- **Gridlines (`plt.grid()`)**: Adds a grid to improve readability.

Output: This enhanced histogram will show bins colored in sky blue, with a black outline around each bin and a grid for better readability.

Why Use Histograms?

- **Data Distribution:** Histograms are great for visualizing the distribution of a dataset (e.g., how test scores are distributed).
- **Binning:** You can adjust the number of bins to see more or less detail in the distribution.

Common Customizations for Histograms:

1. **Number of Bins (`bins`)**: Control how many intervals (bins) the data is split into.
2. **Color (`color`)**: Change the color of the bars.
3. **Edge Color (`edgecolor`)**: Add a border color around each bar.
4. **Transparency (`alpha`)**: Adjust the transparency of the bars to make overlapping bins easier to see.
5. **Gridlines (`plt.grid()`)**: Add gridlines for better readability.
6. **Axis Labels and Title (`xlabel`, `ylabel`, `title`)**: Customize the labels and title.
7. **Orientation (`plt.hist(orientation='horizontal')`)**: Display the histogram horizontally instead of vertically.

Here are **3 sample datasets** and **3 questions for each type of graph** (total 12 questions) to help you practice and check your understanding of line plots, bar charts, histograms, and combined bar/line charts.

Sample Dataset 1: Test Scores of Students

```
students = ['Alice', 'Bob', 'Charlie', 'David', 'Eva']
scores = [85, 92, 78, 88, 95]
improvement = [2, 3, -1, 1, 5]
```

Line Plot Questions:

1. Create a **line plot** to show how the **scores** of students are distributed. Use labels and titles.
2. Plot both **scores** and **improvement** on the same graph using different line styles and colors. Ensure both lines are easily distinguishable.

3. Annotate the plot to indicate which student had the **highest improvement** in scores.

Sample Dataset 2: Monthly Sales Data

```
months = ['January', 'February', 'March', 'April', 'May']
sales = [2500, 3000, 2200, 2800, 3200]
expenses = [1800, 1500, 1900, 2100, 2000]
```

Bar Chart Questions:

1. Create a **bar chart** that shows the **sales** for each month. Add the sales values above each bar.
2. Compare the **sales** and **expenses** data using **two separate bar charts** in the same figure (side by side or stacked).
3. Enhance the chart by using different colors for **sales** and **expenses**, and add a legend to explain which bar represents what.

Sample Dataset 3: Test Scores Distribution

```
test_scores = [55, 60, 65, 70, 75, 75, 80, 85, 90, 95, 100, 100, 100,
70, 85, 90, 60, 65, 95, 80]
```

Histogram Questions:

1. Create a **histogram** to display the distribution of the **test scores**. Choose an appropriate number of bins.
2. Add **customizations** to your histogram, such as different colors and gridlines, to make it more visually appealing.
3. Analyze the **distribution of test scores** by adjusting the number of bins. How does the interpretation change as you increase or decrease the number of bins?

Sample Dataset 4: Product Sales and Profit

```
products = ['Product A', 'Product B', 'Product C', 'Product D']
sales = [4000, 3000, 5000, 4500]
profit = [600, 700, 800, 750]
```

Combined Bar/Line Chart Questions:

1. Create a **bar chart** for **sales** and plot a **line chart** on top of it for **profit**. Ensure both sales and profit are clearly visible.
2. Add the **sales values** over each bar and the **profit values** over each data point on the line. Make sure they are easy to read.
3. Use **different colors** for the bars and the line, and add a **legend** to distinguish between them.

Summary of Questions:

Line Plot (Dataset 1: Test Scores of Students)

1. Create a line plot to show the distribution of student scores.
2. Plot both scores and improvement on the same graph using different line styles.
3. Annotate the student with the highest improvement.

Bar Chart (Dataset 2: Monthly Sales Data)

1. Create a bar chart showing sales per month and add values over each bar.
2. Compare sales and expenses using two bar charts in one figure.
3. Enhance the chart with different colors and a legend.

Histogram (Dataset 3: Test Scores Distribution)

1. Create a histogram to display the test scores distribution.
2. Customize the histogram with colors and gridlines.
3. Analyze how the interpretation changes when adjusting the number of bins.

Combined Bar/Line Chart (Dataset 4: Product Sales and Profit)

1. Create a bar chart for sales and a line chart for profit on the same graph.
2. Add values over the bars and line data points for better readability.
3. Use different colors for bars and lines, and add a legend.

How to Practice:

1. Use the provided datasets to create each type of plot.
2. Answer each of the sample questions by applying what you've learned.
3. Try adding your own customizations to enhance the charts and plots.

Let me know if you need help with any of these tasks or have further questions!

Question Dataset 1: Test Scores of Students

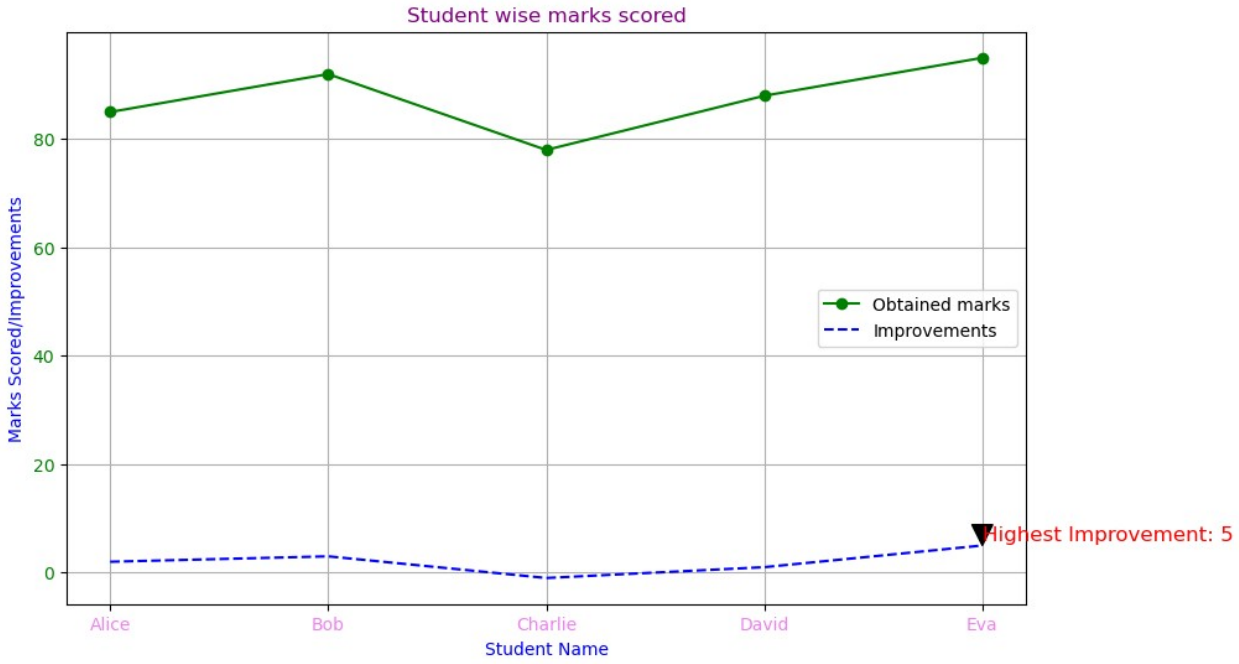
```
students = ['Alice', 'Bob', 'Charlie', 'David', 'Eva']
scores = [85, 92, 78, 88, 95]
improvement = [2, 3, -1, 1, 5]
```

Line Plot Questions:

1. Create a **line plot** to show how the **scores** of students are distributed. Use labels and titles.
2. Plot both **scores** and **improvement** on the same graph using different line styles and colors. Ensure both lines are easily distinguishable.
3. Annotate the plot to indicate which student had the **highest improvement** in scores.

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

```
students = ['Alice', 'Bob', 'Charlie', 'David', 'Eva']
scores = [85, 92, 78, 88, 95]
improvement = [2, 3, -1, 1, 5]

# Set the figure size
plt.figure(figsize=(10,6))

# Plot the scores and improvements
plt.plot(students, scores, color='green', marker='o', label='Obtained
marks')
plt.plot(students, improvement, color='blue', linestyle='--',
label='Improvements')

# Add labels and title
plt.xlabel("Student Name", color='blue')
plt.ylabel("Marks Scored/Improvements", color='blue')
plt.title("Student wise marks scored", color='purple')
plt.grid(True)

# Customize tick colors
plt.xticks(color='violet')
plt.yticks(color='green')

# Add legend
plt.legend()

# Find the student with the highest improvement
max_improvement = max(improvement)  # Find max improvement value
max_index = improvement.index(max_improvement)  # Find the index of
the max improvement

# Annotate the plot with the student who had the highest improvement
plt.annotate(f'Highest Improvement: {max_improvement}',
             xy=(students[max_index], improvement[max_index]),
             xytext=(students[max_index], improvement[max_index] + 1),
# Adjust position of text
             arrowprops=dict(facecolor='black', shrink=0.05),  # Arrow
pointing to the data point
             fontsize=12, color='red')

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



Sample Dataset 2: Monthly Sales Data

```
months = ['January', 'February', 'March', 'April', 'May']
sales = [2500, 3000, 2200, 2800, 3200]
expenses = [1800, 1500, 1900, 2100, 2000]
```

Bar Chart Questions:

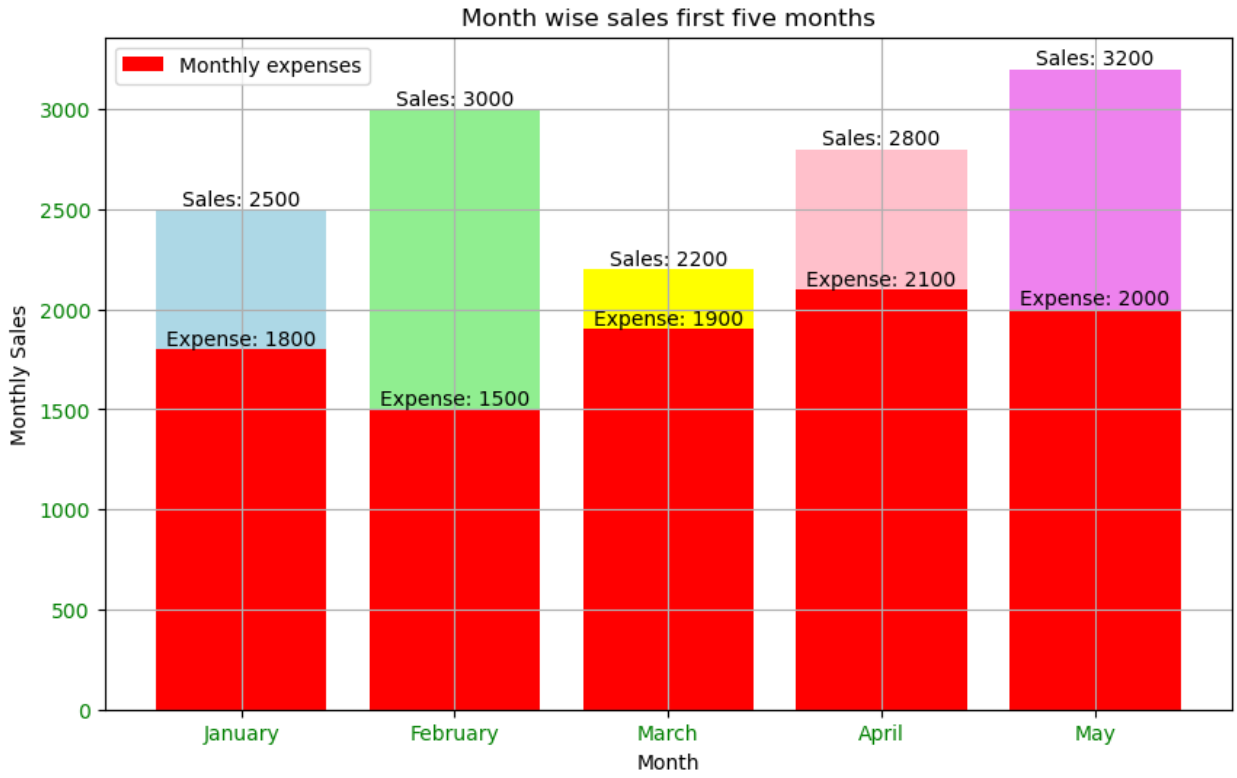
1. Create a **bar chart** that shows the **sales** for each month. Add the sales values above each bar.
2. Compare the **sales** and **expenses** data using **two separate bar charts** in the same figure (side by side or stacked).
3. Enhance the chart by using different colors for **sales** and **expenses**, and add a legend to explain which bar represents what.

Type 1: Staked Bar Chart

```
import matplotlib.pyplot as plt

months = ['January', 'February', 'March', 'April', 'May']
sales = [2500, 3000, 2200, 2800, 3200]
expenses = [1800, 1500, 1900, 2100, 2000]

plt.figure(figsize=(10,6))

sales_graph =
plt.bar(months,sales,color=['lightblue','lightgreen','yellow','pink','
violet'])

for sale in sales_graph:
```

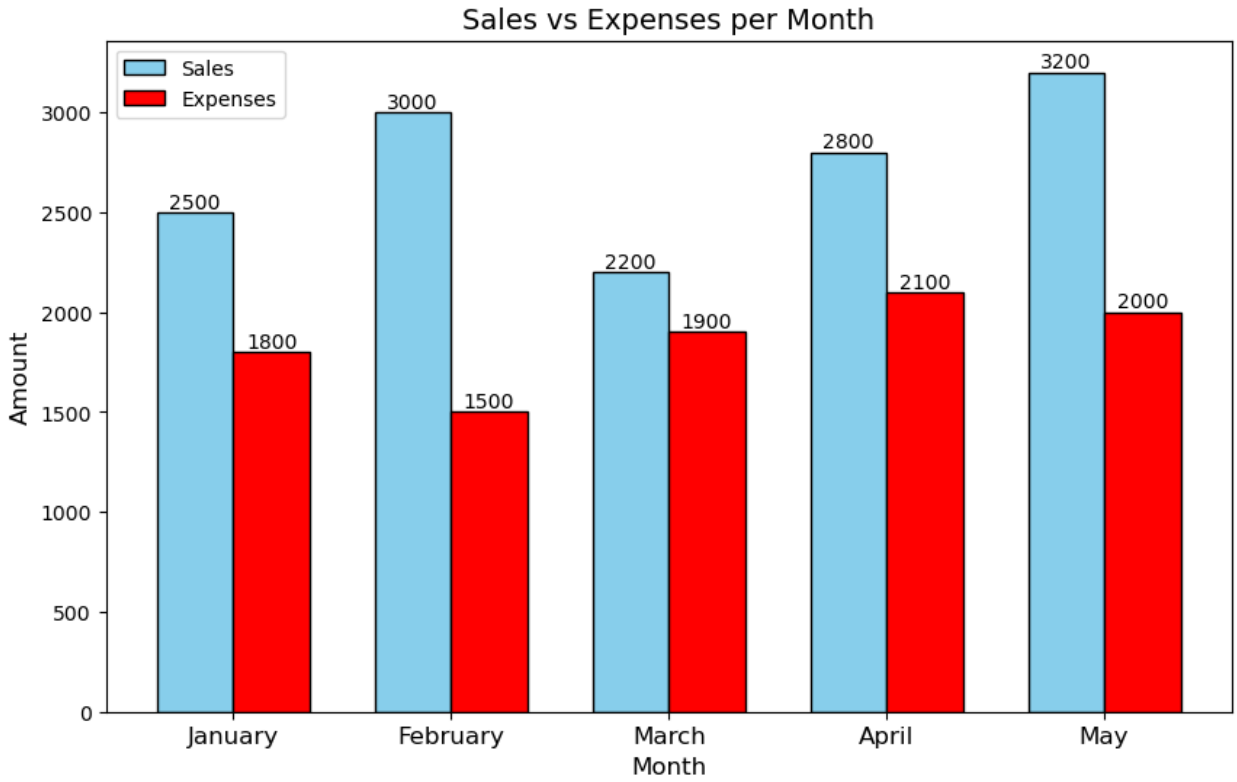
```
    num = sale.get_height()
    plt.text(sale.get_x() + sale.get_width()/2, num,f'Sales:
{num}',ha='center',va='bottom')

expense_graph = plt.bar(months,expenses,label='Monthly
expenses',color='red')
for expense in expense_graph:
    num1 = expense.get_height()
    plt.text(expense.get_x() + expense.get_width()/2,num1,f'Expense:
{num1}',ha='center',va='bottom')


plt.xlabel("Month")
plt.ylabel("Monthly Sales")
plt.title("Month wise sales first five months")

plt.xticks(color='green',fontsize=10)
plt.yticks(color='green',fontsize=10)


plt.grid(True)
plt.legend(loc='upper left')
```

```
<matplotlib.legend.Legend at 0x196763f05c0>
```



Type 2: Side by side bar chart

```
import matplotlib.pyplot as plt

months = ['January', 'February', 'March', 'April', 'May']
sales = [2500, 3000, 2200, 2800, 3200]
expenses = [1800, 1500, 1900, 2100, 2000]

# Set figure size
plt.figure(figsize=(10,6))

# Define the width of each bar
bar_width = 0.35

# Positioning for the bars
r1 = range(len(months))  # Position of the sales bars
r2 = [x + bar_width for x in r1]  # Position of the expenses bars

# Create the bar chart for sales
sales_graph = plt.bar(r1, sales, width=bar_width, color='skyblue',
edgecolor='black', label='Sales')

# Create the bar chart for expenses
expense_graph = plt.bar(r2, expenses, width=bar_width, color='red',
edgecolor='black', label='Expenses')

# Adding the values on top of each bar
for bar in sales_graph:
    plt.text(bar.get_x() + bar.get_width()/2, bar.get_height(),
f'{bar.get_height()}', ha='center', va='bottom')

for bar in expense_graph:
    plt.text(bar.get_x() + bar.get_width()/2, bar.get_height(),
f'{bar.get_height()}', ha='center', va='bottom')

# Add labels and title
plt.xlabel('Month', fontsize=12)
plt.ylabel('Amount', fontsize=12)
plt.title('Sales vs Expenses per Month', fontsize=14)

# Add custom tick marks for both bars
plt.xticks([r + bar_width/2 for r in r1], months, fontsize=12)

# Add legend
plt.legend()

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



Sample Dataset 3: Test Scores Distribution

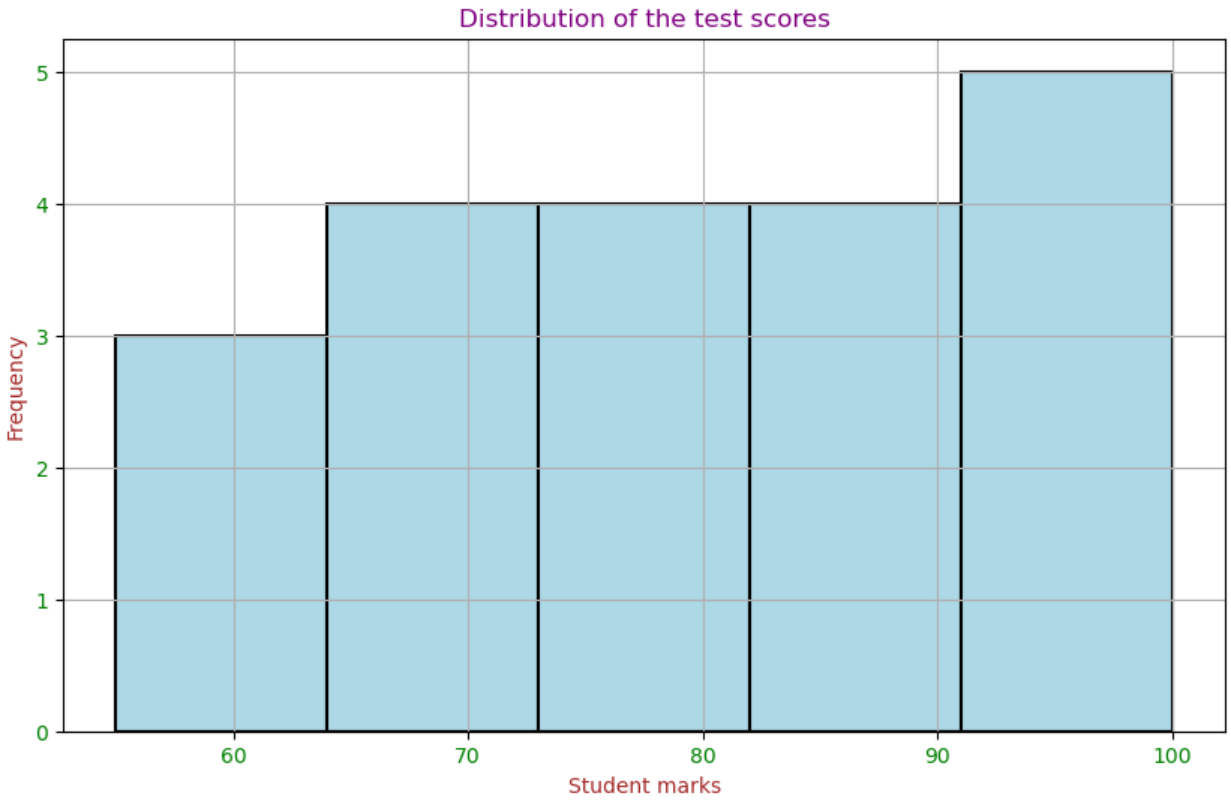
```
test_scores = [55, 60, 65, 70, 75, 75, 80, 85, 90, 95, 100, 100, 100,
70, 85, 90, 60, 65, 95, 80]
```

Histogram Questions:

1. Create a **histogram** to display the distribution of the **test scores**. Choose an appropriate number of bins.
2. Add **customizations** to your histogram, such as different colors and gridlines, to make it more visually appealing.
3. Analyze the **distribution of test scores** by adjusting the number of bins. How does the interpretation change as you increase or decrease the number of bins?

```
import matplotlib.pyplot as plt

test_scores = [55, 60, 65, 70, 75, 75, 80, 85, 90, 95, 100, 100, 100,
70, 85, 90, 60, 65, 95, 80]

plt.figure(figsize=(10,6))

hist_plot =
plt.hist(test_scores,bins=5,color='lightblue',edgecolor='black',linewi
dth=1.5)
plt.xlabel("Student marks",color='Brown')
plt.ylabel("Frequency",color='Brown')
```

```
plt.title("Distribution of the test scores",color='Purple')

plt.xticks(color='green')
plt.yticks(color='green')

plt.grid(True)
```

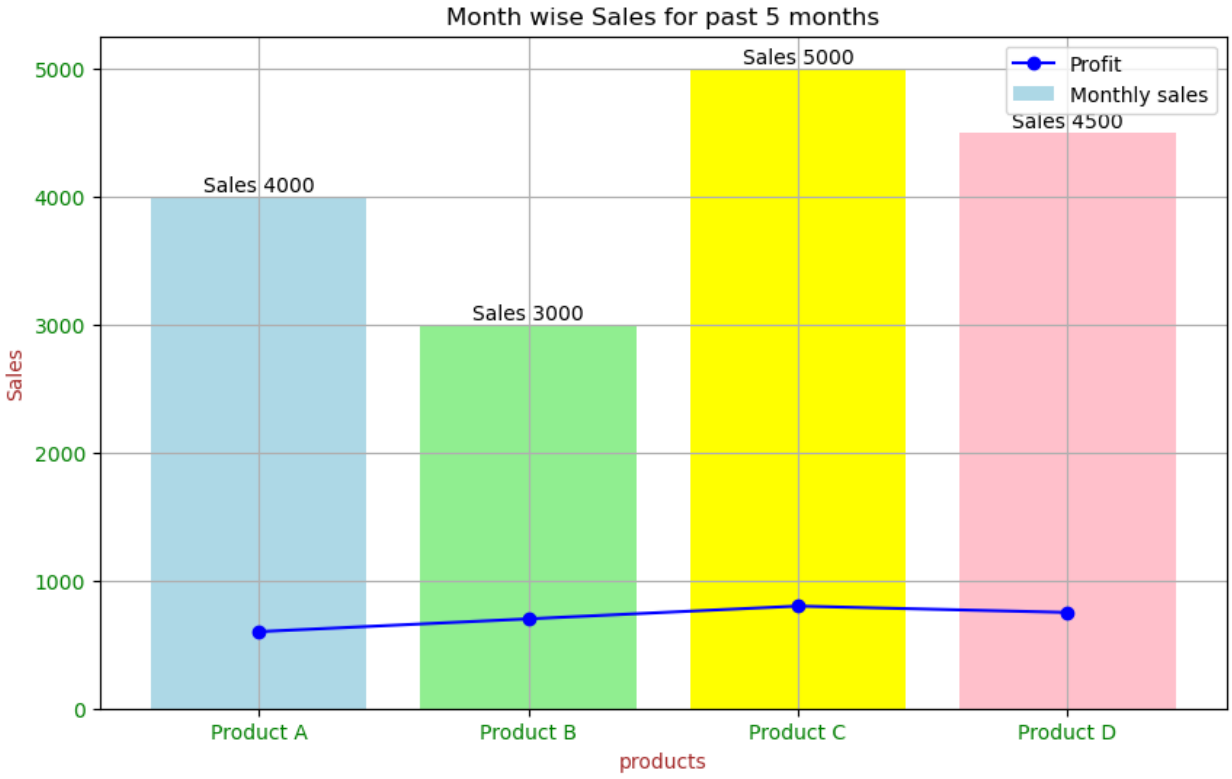


Combined Bar/Line Chart (Dataset 4: Product Sales and Profit)

1. Create a bar chart for sales and a line chart for profit on the same graph.
2. Add values over the bars and line data points for better readability.
3. Use different colors for bars and lines, and add a legend.

```
import matplotlib.pyplot as plt

products = ['Product A', 'Product B', 'Product C', 'Product D']
sales = [4000, 3000, 5000, 4500]
profit = [600, 700, 800, 750]

plt.figure(figsize=(10,6))

sales_bar =
plt.bar(products,sales,color=['lightblue','lightgreen','yellow','pink'
,'violet'],label='Monthly sales')

for value in sales_bar:
```

```
    height=value.get_height()
    plt.text(value.get_x() + value.get_width()/2,height,f'Sales
{height}',ha='center',va='bottom')

line_chart =
plt.plot(products,profit,color='blue',label='Profit',marker='o')

plt.xlabel('products',color='Brown')
plt.ylabel('Sales',color='Brown')
plt.title('Month wise Sales for past 5 months')
plt.grid(True)

plt.xticks(color='green')
plt.yticks(color='green')

plt.legend()

<matplotlib.legend.Legend at 0x19677ea2810>
```



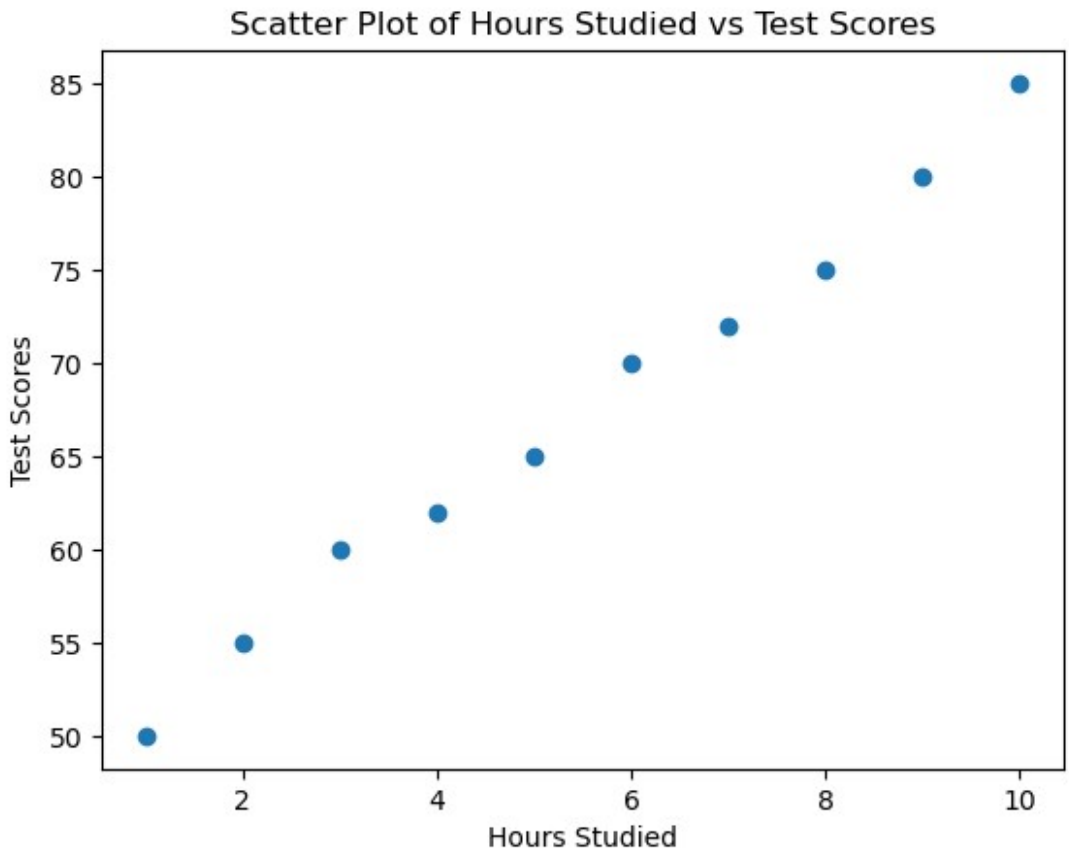
4. Scatter Plot

A **scatter plot** is used to display the relationship between two continuous variables. Each point on the plot represents a pair of values for the two variables, allowing you to see if there is any correlation or pattern.

Step 1: Simple Scatter Plot Example

Let's create a basic scatter plot showing the relationship between **hours studied** and **test scores**.

```
import matplotlib.pyplot as plt

# Data for the scatter plot
hours_studied = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
test_scores = [50, 55, 60, 62, 65, 70, 72, 75, 80, 85]

# Create the scatter plot
plt.scatter(hours_studied, test_scores)

# Add labels and a title
plt.xlabel('Hours Studied')
plt.ylabel('Test Scores')
plt.title('Scatter Plot of Hours Studied vs Test Scores')

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



Explanation:

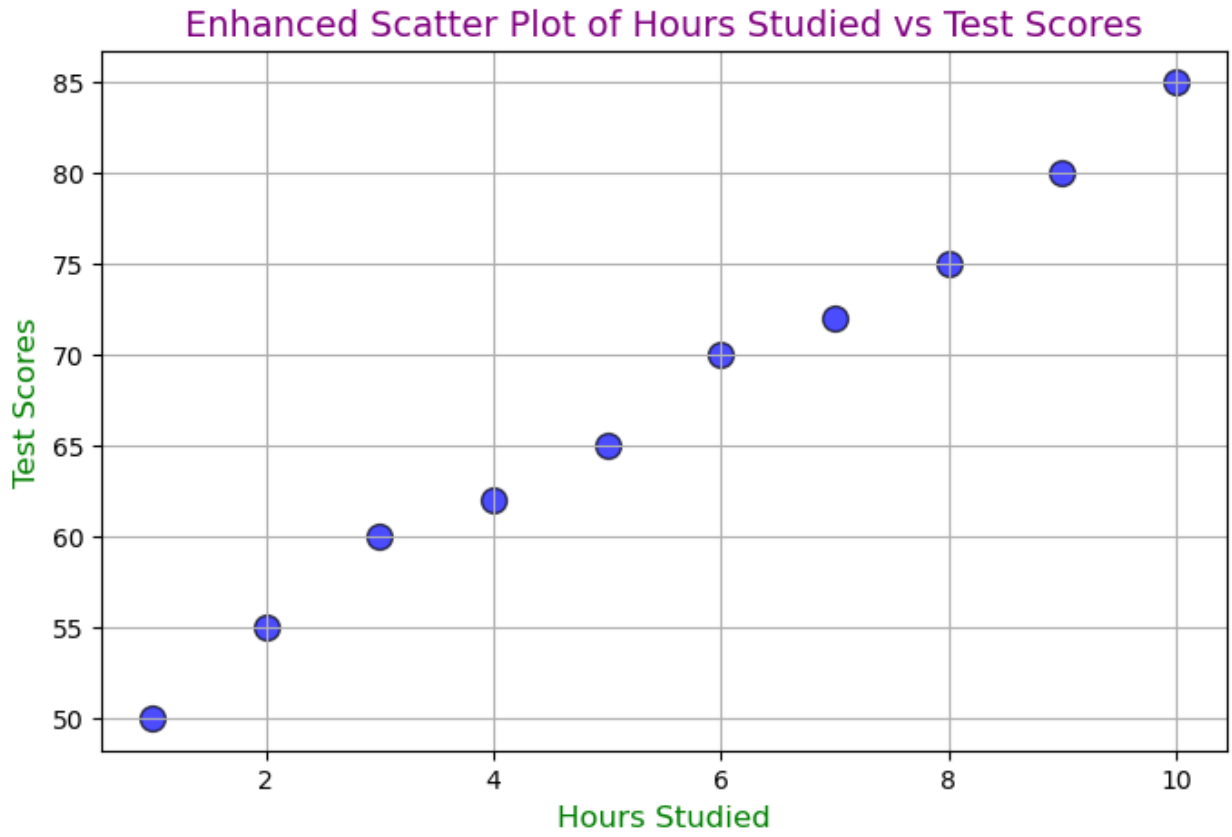
- **`hours_studied`**: The x-axis represents the number of hours studied.
- **`test_scores`**: The y-axis represents the corresponding test scores.
- **`plt.scatter()`**: This function creates the scatter plot, where each point represents a student's hours studied and their corresponding test score.

Output: You'll see a scatter plot with points that show how studying hours relate to test scores.

Step 2: Enhancing the Scatter Plot

Now let's make the scatter plot more visually appealing by customizing the **color**, **marker size**, **adding a grid**, and **annotating points**.

```
import matplotlib.pyplot as plt

# Data for the scatter plot
hours_studied = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
test_scores = [50, 55, 60, 62, 65, 70, 72, 75, 80, 85]

# Set the figure size
plt.figure(figsize=(8, 5))

# Create an enhanced scatter plot
plt.scatter(hours_studied, test_scores,
            color='blue',         # Color of points
            s=100,                # Size of points
            edgecolor='black',     # Add a black edge around points
            alpha=0.7)            # Set transparency level

# Add labels and a title
plt.xlabel('Hours Studied', fontsize=12, color='green')
plt.ylabel('Test Scores', fontsize=12, color='green')
plt.title('Enhanced Scatter Plot of Hours Studied vs Test Scores',
fontsize=14, color='purple')

# Add gridlines
plt.grid(True)

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



Enhancements Explained:

- `color='blue'`: Changes the color of the points to blue.
- `s=100`: Increases the size of the points (default size is 20).
- `edgecolor='black'`: Adds a black edge around each point for better visibility.
- `alpha=0.7`: Adds transparency to the points, which can help make overlapping points more readable.
- **Gridlines**: The grid improves readability of the relationship between the variables.

Output: This enhanced scatter plot will show larger blue points with black edges and transparency. The grid and labels will help make the plot more informative and visually appealing.

Why Use Scatter Plots?

- **Correlation Analysis**: Scatter plots are perfect for showing relationships between two continuous variables. You can visually inspect if there's a correlation (positive, negative, or none).
- **Outlier Detection**: Scatter plots make it easy to spot outliers in the data.

Common Customizations for Scatter Plots:

1. **Point Size (`s`)**: Control the size of each data point.
2. **Point Color (`color`)**: Set the color of the points.
3. **Transparency (`alpha`)**: Set transparency to make overlapping points easier to distinguish.
4. **Point Shape (`marker`)**: Customize the shape of points (e.g., circles, squares, triangles).
5. **Annotations (`plt.annotate()`)**: Add text annotations to specific points.
6. **Gridlines (`plt.grid()`)**: Add gridlines for better readability.
7. **Axis Labels and Title (`xlabel`, `ylabel`, `title`)**: Customize the labels and title with fonts and colors.
8. **Edge Color (`edgecolor`)**: Add a colored border to each point.

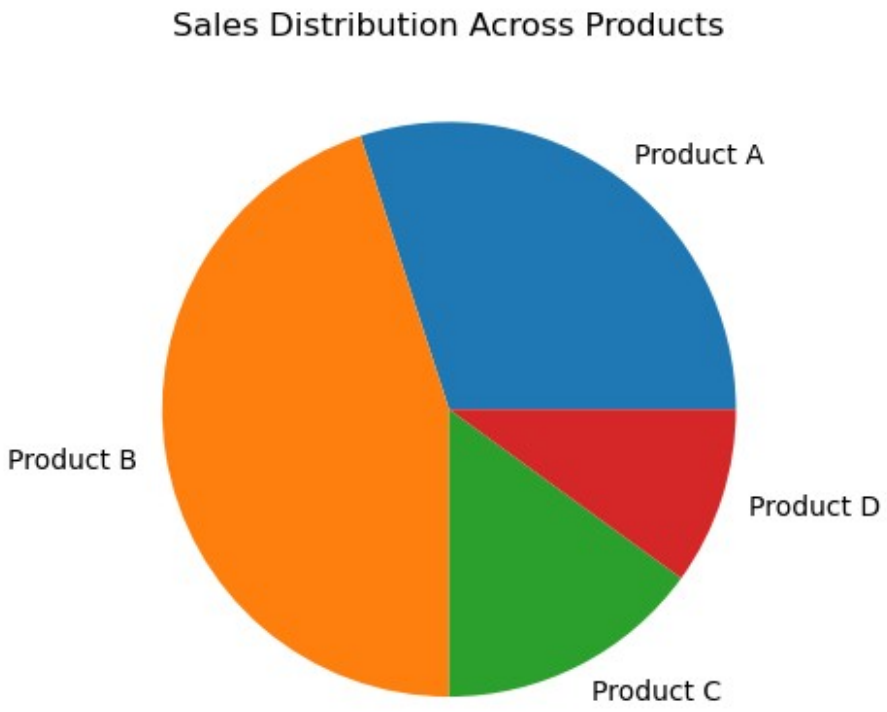
5. Pie Chart

A **pie chart** is a circular chart divided into sectors, where each sector represents a proportion of the total. Pie charts are most useful for showing the relative sizes of parts to a whole.

Step 1: Simple Pie Chart Example

Let's start by creating a basic pie chart that shows the distribution of sales in different product categories.

```
import matplotlib.pyplot as plt

# Data for the pie chart
categories = ['Product A', 'Product B', 'Product C', 'Product D']
sales = [300, 450, 150, 100]

# Create the pie chart
plt.pie(sales, labels=categories)

# Add a title
plt.title('Sales Distribution Across Products')

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



Explanation:

- **`sales`**: The data represents the sales figures for different products.
- **`categories`**: These are the labels corresponding to each product.
- **`plt.pie()`**: This function creates the pie chart, using `sales` for the sizes of the slices and `categories` as the labels.

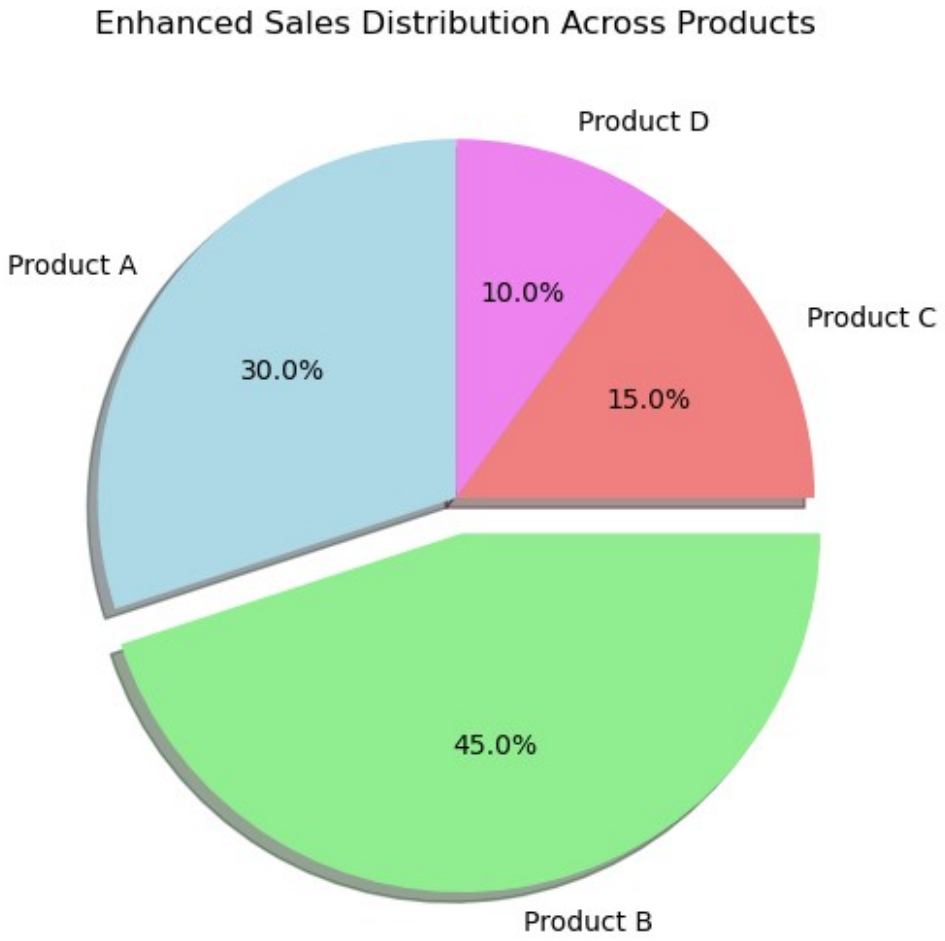
Output: You'll see a basic pie chart with four slices representing sales for the four product categories.

Step 2: Enhancing the Pie Chart

Now, let's make the pie chart more visually appealing by adding colors, a percentage display, exploding one slice, and adding a shadow.

```
import matplotlib.pyplot as plt

# Data for the pie chart
categories = ['Product A', 'Product B', 'Product C', 'Product D']
sales = [300, 450, 150, 100]

plt.figure(figsize=(10,6))  # set chart size
# Create an enhanced pie chart
plt.pie(sales,
        labels=categories,
```

```
        colors=['lightblue', 'lightgreen', 'lightcoral', 'violet'], # 
Set colors for each slice
        autopct='%1.1f%%',   # Display percentages
        explode=(0, 0.1, 0, 0),  # 'Explode' the 2nd slice
        shadow=True,         # Add a shadow
        startangle=90)       # Start the first slice at 90 degrees

# Add a title
plt.title('Enhanced Sales Distribution Across Products')

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



Enhancements Explained:

- **`colors=colors`**: Specifies the colors for each slice of the pie.
- **`autopct='%1.1f%%'`**: Displays the percentage value on each slice with one decimal place.
- **`explode=(0, 0.1, 0, 0)`**: "Explodes" (pulls out) the second slice slightly to highlight it.

- `shadow=True`: Adds a shadow for a 3D effect.
- `startangle=90`: Starts the first slice at the top (90 degrees).

Output: This enhanced pie chart will have different colors for each slice, percentage labels, a shadow effect, and an exploded slice to highlight **Product B**.

Why Use Pie Charts?

- **Proportions**: Pie charts are great for showing how much each category contributes to the total.
- **Simple Visualization**: They are easy to understand when showing simple datasets with relatively few categories.

Common Customizations for Pie Charts:

1. **Colors (`colors`)**: Set custom colors for each slice.
2. **Percentage Labels (`autopct`)**: Display percentages on each slice.
3. **Exploding Slices (`explode`)**: Pull out individual slices for emphasis.
4. **Shadows (`shadow`)**: Add a shadow for a 3D effect.
5. **Start Angle (`startangle`)**: Rotate the starting position of the first slice.
6. **Slice Labels (`labels`)**: Add descriptive labels to each slice.
7. **Legend (`plt.legend()`)**: Add a legend to explain each slice.

6. Box Plot

What is a Box Plot?

A **box plot** (also known as a **box-and-whisker plot**) is a standardized way of displaying the distribution of data based on a **five-number summary**:

1. **Minimum**: The smallest data point excluding outliers.
2. **First Quartile (Q1)**: The 25th percentile of the data.
3. **Median (Q2)**: The 50th percentile or middle of the dataset.
4. **Third Quartile (Q3)**: The 75th percentile of the data.
5. **Maximum**: The largest data point excluding outliers.

The **box** represents the interquartile range (IQR) of the data (between Q1 and Q3), and the **whiskers** extend from the box to the minimum and maximum values (excluding outliers). Outliers are typically plotted as individual points beyond the whiskers.

Purpose of a Box Plot:

Box plots are used to:

- **Visualize the spread and skewness of data**: The width of the box shows the spread of the middle 50% of the data (IQR).

- **Identify outliers**: Outliers, which are unusually high or low values, are shown as points outside the whiskers.
- **Compare distributions**: Multiple box plots can be used side by side to compare the distribution of different groups.

Key Terms in a Box Plot:

1. **Median (Q2)**

- The **median** is the middle value in a dataset when it is ordered from lowest to highest.
- It divides the dataset into two equal halves.
- It is also known as the **50th percentile**.

2. **First Quartile (Q1)**

- **Q1** is the **25th percentile**: 25% of the data points lie below this value, and 75% are above it.
- It represents the lower bound of the **middle 50%** of the data.

3. **Third Quartile (Q3)**

- **Q3** is the **75th percentile**: 75% of the data points lie below this value, and 25% are above it.
- It represents the upper bound of the **middle 50%** of the data.

4. **Interquartile Range (IQR)**

- The **IQR** is the difference between the third quartile (**Q3**) and the first quartile (**Q1**): [IQR = Q3 - Q1]
- The **IQR** measures the spread of the middle 50% of the data and is a useful measure of variability.

5. **Whiskers**

- The **whiskers** extend from the edges of the box (Q1 and Q3) to the smallest and largest values in the data that are **not considered outliers**.
- The **length of the whiskers** is typically defined as 1.5 times the IQR: [\text{Lower whisker} = Q1 - 1.5 \times IQR] [\text{Upper whisker} = Q3 + 1.5 \times IQR]
- Values outside the range of the whiskers are considered **outliers**.

6. **Outliers**

- Outliers are data points that fall below the lower whisker or above the upper whisker. They are typically represented as individual points beyond the whiskers in a box plot.
- Mathematically, any data point that falls outside the following range is considered an outlier: [\text{Outliers} < Q1 - 1.5 \times IQR \quad \text{or} \quad \text{Outliers} > Q3 + 1.5 \times IQR]

Structure of a Box Plot:

- **The Box**: The box extends from Q1 (25th percentile) to Q3 (75th percentile), and the line inside the box represents the median (Q2 or 50th percentile).
- **The Whiskers**: The whiskers extend from Q1 and Q3 to the smallest and largest values within 1.5 times the IQR.

- **Outliers**: Outliers are plotted as individual points beyond the whiskers.

Example of How to Calculate Q1, Q2, Q3, and IQR:

Let's use an example dataset to understand how to calculate the quartiles:

```
data = [25, 30, 35, 40, 45, 50, 55, 60, 65, 70, 75, 80, 85]
```

1. **Step 1: Order the Data (Ascending/Descending)**
   The data is already in order.
2. **Step 2: Calculate the Median (Q2)**
   The median is the middle value:
   - For this dataset, the median is **55** (the 7th value).
3. **Step 3: Calculate Q1 and Q3**
   - **Q1 (25th percentile)**: The median of the lower half of the data (excluding the overall median): [Q1 = 40]
   - **Q3 (75th percentile)**: The median of the upper half of the data: [Q3 = 70]
4. **Step 4: Calculate the Interquartile Range (IQR)**
   [IQR = Q3 - Q1 = 70 - 40 = 30]
5. **Step 5: Determine Whiskers and Outliers**
   - Lower Whisker:
     [Q1 - 1.5 \times IQR = 40 - 1.5 \times 30 = -5] The minimum value in the dataset is **25**, which is within this range.
   - Upper Whisker:
     [Q3 + 1.5 \times IQR = 70 + 1.5 \times 30 = 115] The maximum value in the dataset is **85**, which is also within this range.
   - There are no outliers in this dataset.

Summary of Key Concepts:

Term	Definition
Q1	25th percentile (first quartile)
Q2	50th percentile (median)
Q3	75th percentile (third quartile)
IQR	Interquartile range: the difference between Q3 and Q1
Whiskers	Extend to the minimum and maximum values within 1.5 times the IQR
Outliers	Data points outside the whiskers

When to Use Box Plots:

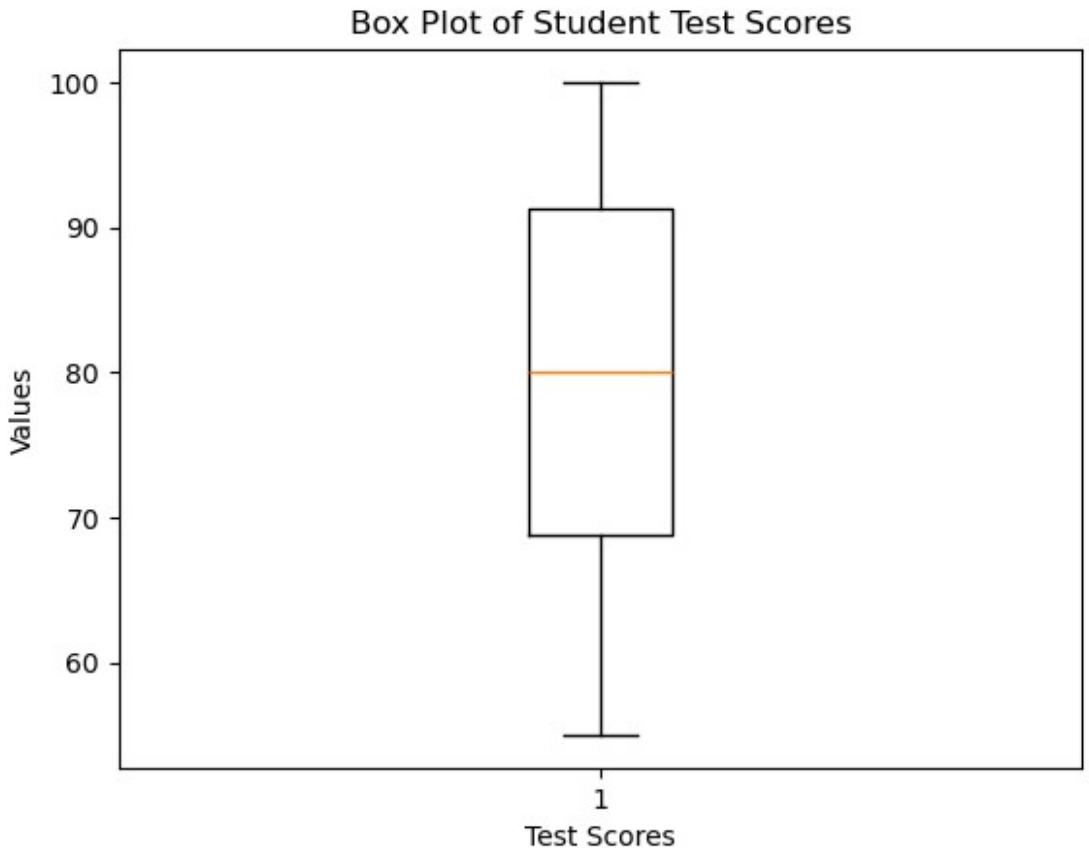
- When you need to visualize the **spread** and **central tendency** of a dataset.
- When you want to compare the distributions of multiple groups.
- When identifying **outliers** is important in your analysis.

A **box plot** displays the distribution of data based on a five-number summary: **minimum**, **first quartile (Q1)**, **median**, **third quartile (Q3)**, and **maximum**. It helps identify the **spread**, **skewness**, and **outliers** in a dataset.

Step 1: Simple Box Plot Example

Let's start by creating a basic box plot to visualize the distribution of test scores for a class of students.

```
import matplotlib.pyplot as plt

# Data for the box plot (test scores of students)
test_scores = [55, 60, 65, 70, 75, 75, 80, 85, 90, 95, 100, 100, 100,
70, 85, 90, 60, 65, 95, 80]

# Create a basic box plot
plt.boxplot(test_scores)

# Add labels and a title
plt.xlabel('Test Scores')
plt.ylabel('Values')
plt.title('Box Plot of Student Test Scores')

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



Explanation:

- **`plt.boxplot()`**: This function generates the box plot, with the box representing the interquartile range (IQR) and the "whiskers" extending to the minimum and maximum values (excluding outliers).
- **Data**: The list `test_scores` represents the dataset.

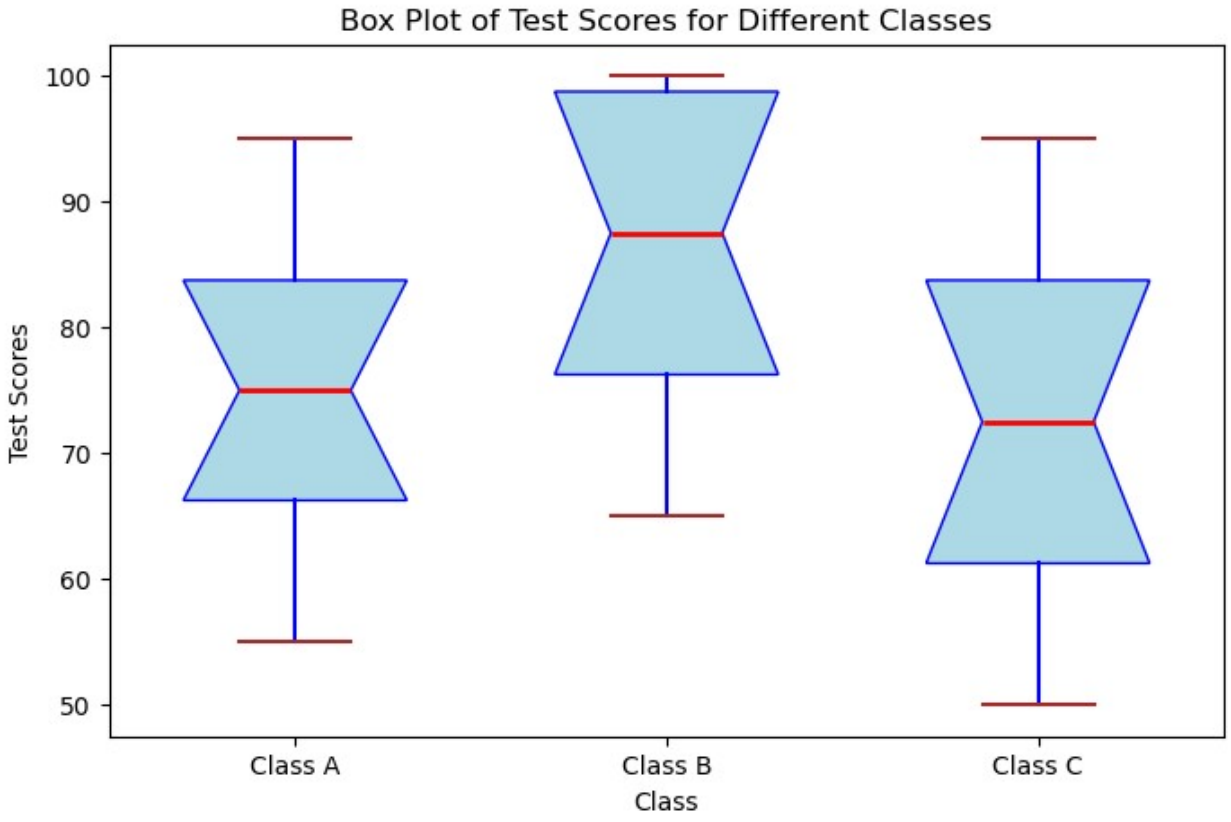
Output: This will produce a simple box plot showing the distribution of the test scores, including the **median** (middle line in the box), the **quartiles**, and possibly some **outliers**.

Step 2: Enhancing the Box Plot

Now, let's make the box plot more informative by adding multiple box plots for different groups (e.g., test scores from different classes) and customizing the appearance (colors, labels, etc.).

```
import matplotlib.pyplot as plt

# Data for multiple groups (test scores from three different classes)
class_A = [55, 60, 65, 70, 75, 75, 80, 85, 90, 95]
class_B = [65, 70, 75, 80, 85, 90, 95, 100, 100, 100]
class_C = [50, 55, 60, 65, 70, 75, 80, 85, 90, 95]
```

```
# Group the data
data = [class_A, class_B, class_C]

# Create an enhanced box plot
plt.figure(figsize=(8, 5))
plt.boxplot(data, patch_artist=True, notch=True, widths=0.6,
            boxprops=dict(facecolor='lightblue', color='blue'),
            medianprops=dict(color='red', linewidth=2),
            whiskerprops=dict(color='blue', linewidth=1.5),
            capprops=dict(color='brown', linewidth=1.5),
            flierprops=dict(marker='o', color='red', markersize=8))

# Add custom labels for the x-axis
plt.xticks([1, 2, 3], ['Class A', 'Class B', 'Class C'])

# Add labels and a title
plt.xlabel('Class')
plt.ylabel('Test Scores')
plt.title('Box Plot of Test Scores for Different Classes')

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



Enhancements Explained:

- **Multiple Box Plots**: We created three box plots, one for each class (`class_A`, `class_B`, `class_C`).
- **`patch_artist=True`**: Fills the boxes with color.
- **`notch=True`**: Adds notches to indicate the confidence interval around the median.
- **Custom Properties**:
  - **`boxprops`**: Customizes the appearance of the box (light blue fill, blue edge).
  - **`medianprops`**: Sets the color and width of the median line (red).
  - **`whiskerprops`**: Customizes the appearance of the whiskers (blue color, thicker lines).
  - **`capprops`**: Adjusts the caps at the end of the whiskers (blue, thicker lines).
  - **`flierprops`**: Customizes the appearance of outliers (red markers, larger size).
- **`plt.xticks()`**: Customizes the labels on the x-axis to show class names.

Output: This enhanced box plot will show the distribution of test scores for three different classes, with each box colored and labeled accordingly.

Why Use Box Plots?

- **Distribution Analysis**: Box plots help visualize the spread and skewness of data, and they highlight important statistics like the median and quartiles.
- **Outlier Detection**: Box plots easily identify outliers as points that fall outside the whiskers.
- **Comparison**: You can use multiple box plots to compare distributions across different groups.

Common Customizations for Box Plots:

1. **Multiple Box Plots**: Display multiple box plots for different groups side by side.
2. **Colors (`patch_artist=True`)**: Fill the boxes with color to distinguish between different groups.
3. **Notches (`notch=True`)**: Add notches to show the confidence interval around the median.
4. **Box Properties (`boxprops`)**: Customize the color, border, and fill of the boxes.
5. **Whisker and Cap Properties (`whiskerprops`, `capprops`)**: Control the appearance of the whiskers and caps.
6. **Outliers (`flierprops`)**: Change the shape, color, and size of outlier points.
7. **Median Line (`medianprops`)**: Customize the appearance of the median line.
8. **Axis Labels and Title (`xlabel`, `ylabel`, `title`)**: Customize the labels and title with fonts and colors.

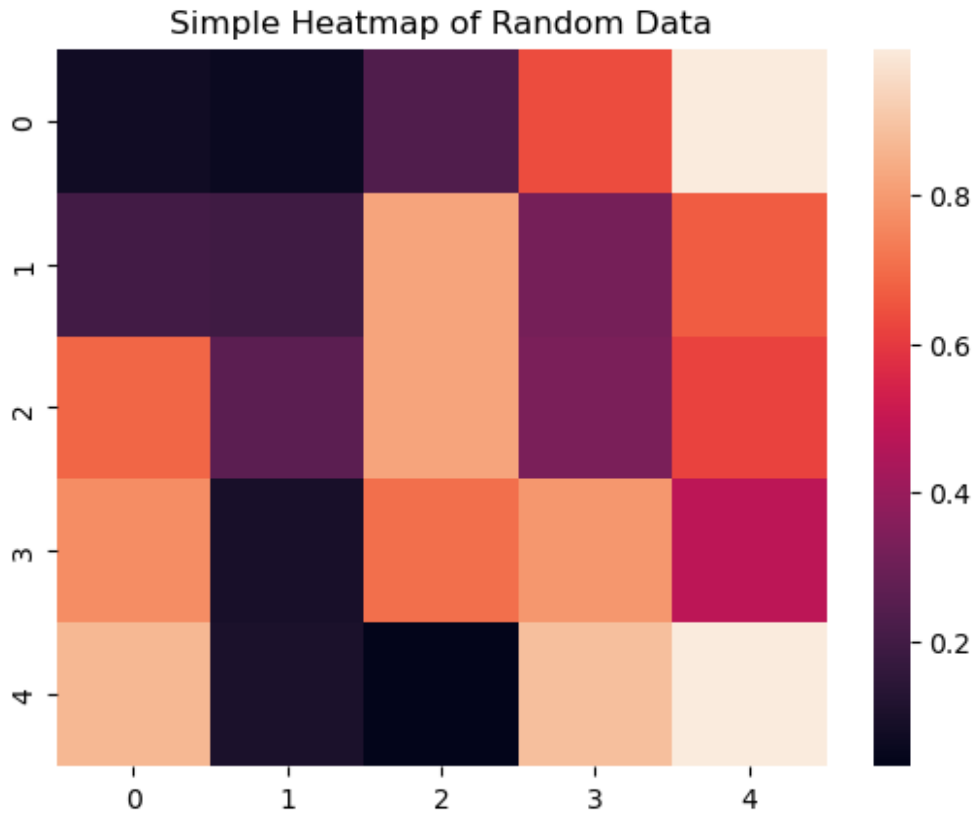
7. Heatmap

A **heatmap** is a graphical representation of data where individual values are represented as colors in a matrix. It's particularly helpful for visualizing the magnitude of values across a two-dimensional grid and is widely used for displaying correlations between features in a dataset.

Step 1: Simple Heatmap Example

Let's start by creating a basic heatmap that shows the correlation between different variables in a dataset.

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

# Data for the heatmap (a 5x5 matrix of random numbers)
data = np.random.rand(5, 5)

# Create a simple heatmap
sns.heatmap(data)

# Add a title
plt.title('Simple Heatmap of Random Data')

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



Explanation:

- `np.random.rand(5, 5)`: Generates a 5x5 matrix of random numbers between 0 and 1.
- `sns.heatmap()`: Creates the heatmap based on the data.
- `plt.title()`: Adds a title to the heatmap.

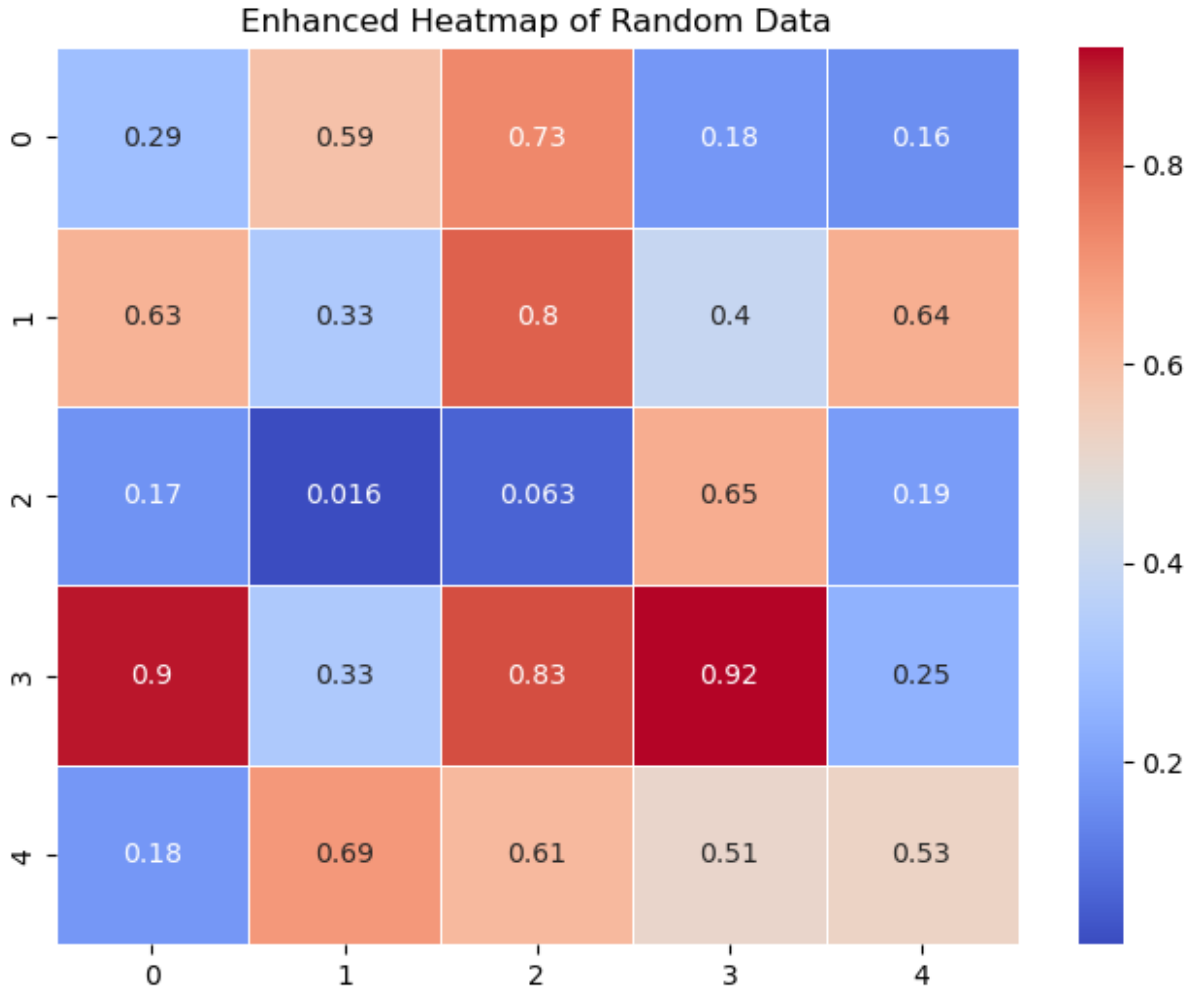
Output: This will produce a simple heatmap where each cell in the matrix is colored based on the value it represents, with a color gradient ranging from lower values to higher values.

Step 2: Enhancing the Heatmap

Let's make the heatmap more informative by adding **annotations**, **custom color maps**, and **a color bar**.

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

# Data for the heatmap (5x5 matrix of random numbers)
data = np.random.rand(5, 5)

# Set the figure size
```

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

# Create an enhanced heatmap with annotations and customizations
sns.heatmap(data,
            annot=True,         # Annotate the cells with the data
values
            cmap='coolwarm',    # Use the 'coolwarm' color map
            linewidths=0.5,     # Add space between cells
            linecolor='white',  # Set the line color between cells
            cbar=True)          # Display the color bar

# Add a title
plt.title('Enhanced Heatmap of Random Data')

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



Enhancements Explained:

- **`annot=True`**: Adds the actual data values inside each cell of the heatmap.

- **`cmap='coolwarm'`**: Uses the `'coolwarm'` color map (blue for low values and red for high values). You can choose other colormaps like `'viridis'`, `'magma'`, etc.
- **`linewidths=0.5`**: Adds lines between the cells to separate them.
- **`linecolor='white'`**: Colors the lines between cells in white.
- **`cbar=True`**: Adds a color bar on the side to show the mapping of data values to colors.

Output: This enhanced heatmap will show the values inside each cell, with a color gradient that varies from blue to red. The color bar will show the scale of the data.

Why Use Heatmaps?

- **Visualizing Correlations**: Heatmaps are often used to visualize correlations between multiple variables in a dataset (e.g., in a correlation matrix).
- **Identifying Patterns**: They help easily identify areas of higher or lower values, making them useful for detecting patterns in data.
- **Data Exploration**: Heatmaps are valuable in exploratory data analysis (EDA), especially when working with large datasets.

Common Customizations for Heatmaps:

1. **Annotations (`annot=True`)**: Display the actual data values in each cell.
2. **Color Maps (`cmap`)**: Choose from different color maps (`coolwarm`, `viridis`, `plasma`, etc.) to represent the data.
3. **Gridlines (`linewidths`, `linecolor`)**: Add or remove lines between cells, and customize their color and thickness.
4. **Color Bar (`cbar=True`)**: Display a color bar to represent the mapping of values to colors.
5. **Axis Labels and Titles**: Customize the labels and titles for the heatmap, just like other plots.

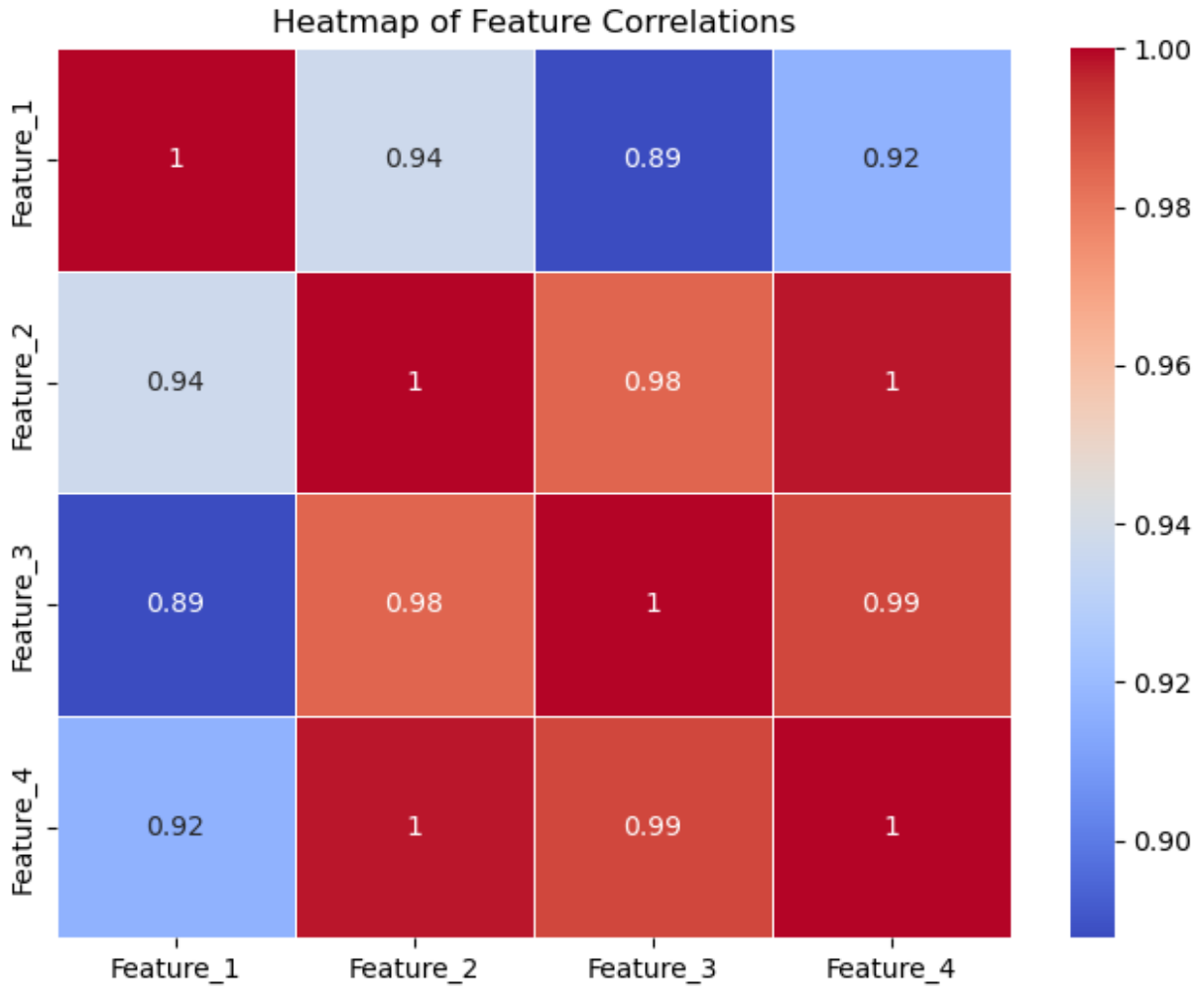
Real-World Use Case of Heatmaps:

Heatmaps are commonly used in **correlation analysis**. Here's an example where a heatmap is used to show the correlation matrix of a dataset.

Example: Heatmap for Correlation Matrix

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

# Sample data
data = {
    'Feature_1': [1, 2, 3, 4, 5, 10],
    'Feature_2': [5, 6, 7, 8, 9, 40],
    'Feature_3': [10, 20, 10, 20, 10,100],
    'Feature_4': [15, 17, 16, 18, 19,80]
```

```
}

# Create a DataFrame
df = pd.DataFrame(data)

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

# Create a heatmap of the correlation matrix
plt.figure(figsize=(8, 6))
sns.heatmap(corr, annot=True,
linewidth=0.6,linecolor='white',cmap='coolwarm')

# Add a title
plt.title('Heatmap of Feature Correlations')

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



Explanation:

- **`df.corr()`**: Computes the correlation matrix of the data.
- **`sns.heatmap(corr, annot=True)`**: Plots the heatmap of the correlation matrix, with the correlation values annotated in each cell.

Output: This heatmap will show the correlations between the features in the dataset, with a color gradient representing the strength of the correlation (e.g., 1 for perfect correlation, 0 for no correlation).

Let's go over your heatmap and focus on interpreting the correlation between **Feature 1** and **Feature 2**, which is `0.94`.

How to Read a Heatmap:

- The heatmap shows a **correlation matrix** between four features (**Feature 1**, **Feature 2**, **Feature 3**, and **Feature 4**) from the dataset.
- The correlation values are displayed within each cell, and the colors represent the strength of the correlation (from blue for negative correlation to red for positive correlation).
- The diagonal is all `1`s because each feature is perfectly correlated with itself.

Understanding the Correlation between Feature 1 and Feature 2 (`0.94`):

The correlation value between **Feature 1** and **Feature 2** is **0.94**, which is a **strong positive correlation**.

This means:

- When **Feature 1** increases, **Feature 2** tends to increase as well.
- Since the correlation is close to `+1`, we can infer that the relationship between **Feature 1** and **Feature 2** is almost linear and very strong.

What this does NOT mean:

- This value only tells you about the **linear relationship**. It doesn't imply causation. A strong correlation doesn't mean one feature causes the other to increase or decrease; it just shows how they move together in a linear fashion.

Why is the Correlation `0.94`?

The correlation is calculated based on how closely the values of **Feature 1** and **Feature 2** follow a linear relationship.

Let’s compare the actual values of **Feature 1** and **Feature 2** from your dataset:

- **Feature 1**: `[1, 2, 3, 4, 5, 10]`
- **Feature 2**: `[5, 6, 7, 8, 9, 40]`

By observing these values:

- Both features are increasing, though **Feature 2** increases more steeply for the final data point.
- **Feature 2** follows a similar linear growth pattern to **Feature 1**, especially for the first five values (`5`, `6`, `7`, `8`, `9`). The last data point (`40`) in **Feature 2** grows much faster compared to **Feature 1**, which increases more gradually (`10`).

This overall similar pattern of growth results in a strong positive correlation, but because the growth is not **perfectly linear** (due to the jump in the last data point), the correlation is **slightly less than `1`** (i.e., `0.94` instead of `1`).

How is Correlation Calculated?

Correlation between two variables (X) and (Y) is calculated using the **Pearson correlation coefficient** formula:

[r = \frac{ \sum (X_i - \overline{X})(Y_i - \overline{Y}) }{ \sqrt{\sum (X_i - \overline{X})^2 \sum (Y_i - \overline{Y})^2 } }]

$$r = \frac{\sum (X_i - \overline{X})(Y_i - \overline{Y})}{\sqrt{\sum (X_i - \overline{X})^2 \sum (Y_i - \overline{Y})^2}}$$

Where:

- (X_i) and (Y_i) are the individual data points for the two features.
- (\overline{X}) and (\overline{Y}) are the means of the two features.
- **r** (the correlation coefficient) will always be between `-1` and `+1`:
  - **r = +1**: Perfect positive correlation.
  - **r = -1**: Perfect negative correlation.
  - **r = 0**: No correlation (random relationship).

In your case, the value `0.94` means that there's a strong, nearly linear relationship between **Feature 1** and **Feature 2**.

Reading the Other Correlations in the Heatmap:

- **Feature 3 and Feature 4 (`0.99`)**: These two features are almost perfectly correlated, meaning that as one increases, the other increases in almost perfect synchrony.
- **Feature 2 and Feature 3 (`0.98`)**: Another strong positive correlation, suggesting a close relationship between these two features.

- **Feature 1 and Feature 3 (`0.89`)**: This is also a strong positive correlation, though slightly weaker than the others.

Takeaways from This Heatmap:

- **Feature 1** and **Feature 2** are **strongly correlated** (`0.94`), indicating they have a similar linear relationship.
- All the features are highly correlated with each other, as all the values in the heatmap are above `0.89`, suggesting that the features move together in similar ways.
- This heatmap is useful for understanding how similar the features are to each other, which can help in **feature selection** or **data reduction**.

Use in EDA:

- You might notice that **Feature 3** and **Feature 4** are almost perfectly correlated (`0.99`). This suggests that one of these features could potentially be removed without losing much information, as they are providing nearly identical information.
- **Identifying relationships** like these helps in refining your dataset before moving to modeling, where highly correlated features might cause **multicollinearity** in certain models.

Summary of Heatmap Key Features:

Feature	Description
Annotations (`annot`)	Display the actual data values inside each cell of the heatmap.
Color Maps (`cmap`)	Customize the color scheme to represent the data (e.g., `'coolwarm'`).
Gridlines (`linewidths`, `linecolor`)	Add lines between the cells and customize their color and thickness.
Color Bar (`cbar`)	Show a color bar to indicate the mapping of data values to colors.
Axis Labels and Titles	Add labels and titles to make the heatmap more informative.

8. Violin Plot

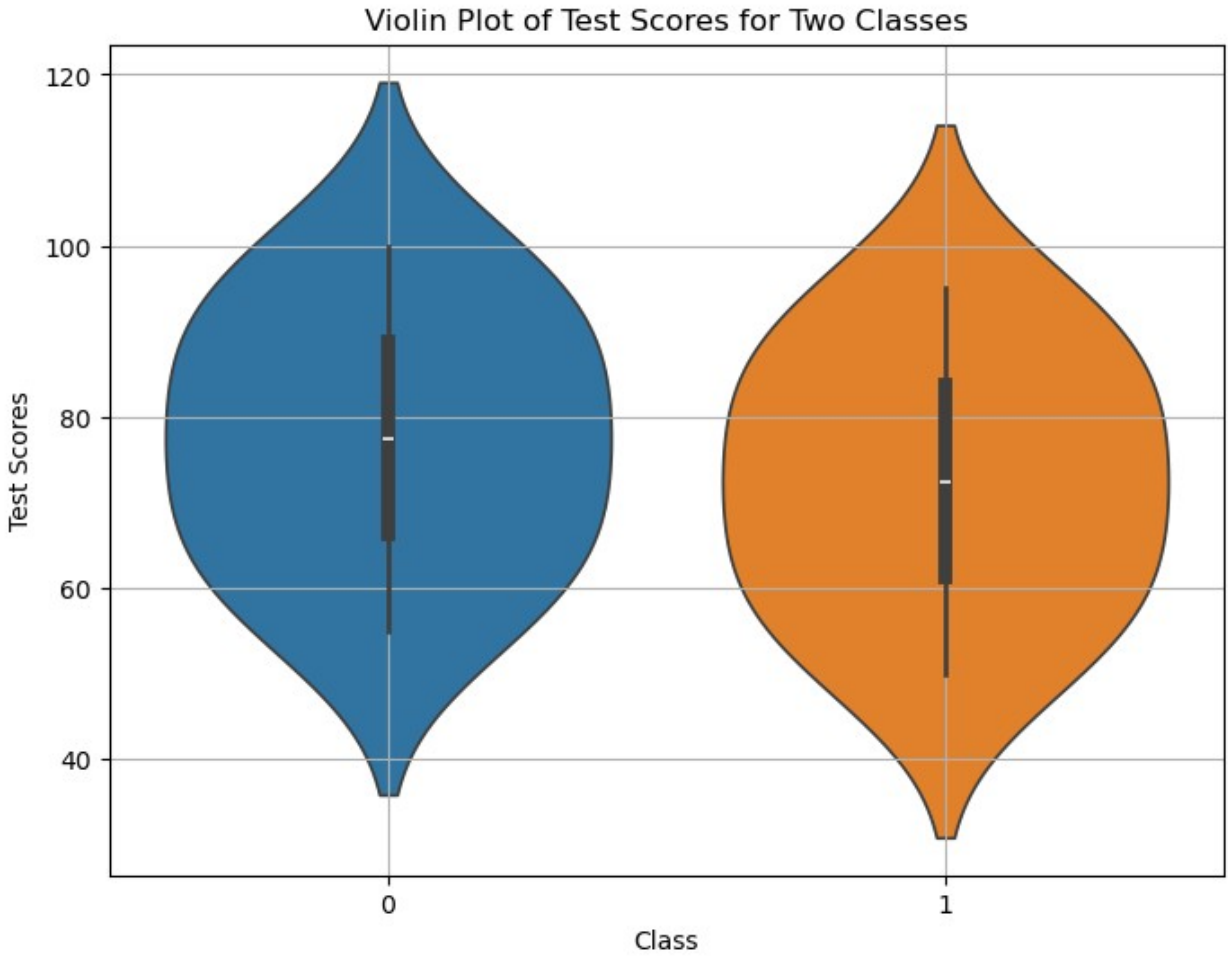
A **violin plot** is similar to a box plot, but it also shows the **kernel density estimation (KDE)** of the data, which gives an idea of the **distribution shape** (how the data is distributed along the x-axis). The width of the violin plot represents the **density** of the data at different values.

Step 1: Simple Violin Plot Example

Let's create a violin plot to visualize the distribution of **test scores** for two different classes of students.

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

```python
# Sample data for two classes
class_A_scores = [55, 60, 65, 70, 75, 80, 85, 90, 95, 100]
class_B_scores = [50, 55, 60, 65, 70, 75, 80, 85, 90, 95]

# Group the data for plotting
data = [class_A_scores, class_B_scores]

# Create the violin plot
plt.figure(figsize=(8, 6))
sns.violinplot(data=data)

# Add labels and a title
plt.xlabel('Class')
plt.ylabel('Test Scores')
plt.title('Violin Plot of Test Scores for Two Classes')
plt.grid(True)

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



Explanation:

- **`sns.violinplot()`**: This function creates the violin plot. The **violin shape** represents the distribution of the data.
- **Data**: We pass the test scores for **Class A** and **Class B** into the `data` parameter, so we get two violins (one for each class).
- **The Shape of the Violin**: The wider the plot at a given score, the more data points exist around that score, indicating **high density**. Conversely, narrow sections represent fewer data points.

Step 2: Enhancing the Violin Plot

Now, let's enhance the violin plot by adding **split violins**, customizing the colors, and using **inner box plots**.

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

# Sample data for three classes
class_A_scores = [55, 60, 65, 70, 75, 80, 85, 90, 95, 100]
class_B_scores = [50, 55, 60, 65, 70, 75, 80, 85, 90, 95]
class_C_scores = [40, 50, 55, 60, 65, 70, 75, 80, 85, 90]

# Create a DataFrame to store the data
import pandas as pd
df = pd.DataFrame({
    'Scores': class_A_scores + class_B_scores + class_C_scores,
    'Class': ['A']*10 + ['B']*10 + ['C']*10
})

# Create an enhanced violin plot
plt.figure(figsize=(8, 6))
sns.violinplot(x='Class', y='Scores', data=df, split=True,
inner="box", palette="Set3")

# Add labels and title
plt.xlabel('Class')
plt.ylabel('Test Scores')
plt.title('Enhanced Violin Plot of Test Scores for Three Classes')

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

```
C:\Users\Dell\AppData\Local\Temp\ipykernel_12888\384898754.py:18:
FutureWarning:

Passing `palette` without assigning `hue` is deprecated and will be
removed in v0.14.0. Assign the `x` variable to `hue` and set
`legend=False` for the same effect.
```

```
  sns.violinplot(x='Class', y='Scores', data=df, split=True,
inner="box", palette="Set3")
```

Enhancements Explained:

- **`split=True`**: When split is set to `True`, the two halves of the violin show the distribution of two different subsets (this is useful when plotting two distributions in one violin, but it requires a `hue` argument).
- **`inner="box"`**: Adds a **box plot** inside the violin to show the median and quartiles. This gives an extra layer of detail about the spread and central tendency of the data.
- **`palette="Set3"`**: Applies a color palette to the violins for better visual distinction.

Why Use a Violin Plot?

- **Combination of Box Plot and KDE**: Violin plots combine the compact summary of a box plot with the detailed information of a KDE. This makes it easier to see both the spread and the shape of the distribution.
- **Comparing Distributions**: Violin plots are great for comparing the distribution of multiple datasets side by side.

- **Understanding Data Density**: The width of the violin at any point tells you how many data points fall in that region, which helps in identifying where most of the data is concentrated.

Key Features of a Violin Plot:

1. **KDE (Kernel Density Estimation)**: Shows the probability density of the data, giving insight into the **shape** and **distribution** (e.g., skewness, multimodality).
2. **Symmetry**: Violin plots are symmetric around the central axis.
3. **Inner Box Plot**: You can add a box plot inside the violin to show **median** and **quartile** values, just like in a regular box plot.
4. **Multiple Distributions**: You can compare distributions for multiple groups side by side.

Real-World Use Case for Violin Plots:

Violin plots are commonly used in **scientific studies** to compare different groups. For instance, in a clinical trial, you might compare the distribution of patient outcomes (e.g., blood pressure) across different treatment groups. The violin plot would show both the **central tendency** and the **variability** in each group, as well as the **density** of data points at different levels.

Common Customizations for Violin Plots:

1. **Splitting Violins (`split=True`)**: Show two distributions within the same violin.
2. **Inner Box Plot (`inner="box"`)**: Add a box plot inside the violin for additional insight.
3. **Colors (`palette`)**: Customize the colors for each violin using a color palette.
4. **Orientation**: You can change the orientation of the violin plot (horizontal or vertical) by switching the axes.

9. Pair Plot

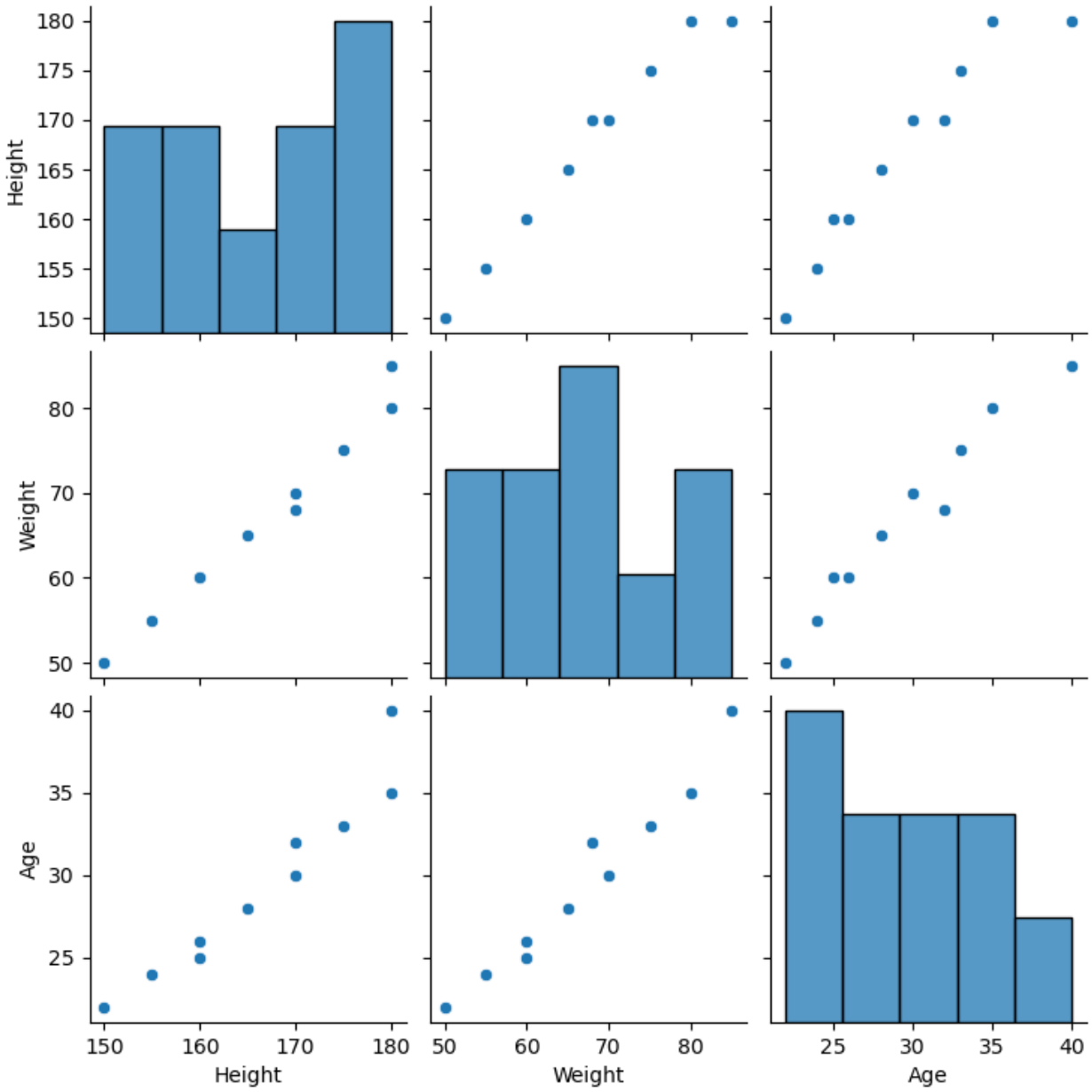
A **pair plot** (also known as a **scatterplot matrix**) is a grid of scatter plots that show the relationships between all pairs of features in a dataset. It also includes histograms (or KDE plots) to show the distribution of each feature on the diagonal.

Step 1: Simple Pair Plot Example

Let's create a pair plot for a dataset with three features: **height**, **weight**, and **age**.

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

# Sample data
import pandas as pd

data = {
    'Height': [160, 170, 150, 180, 165, 155, 175, 160, 170, 180],
```

```
    'Weight': [60, 70, 50, 80, 65, 55, 75, 60, 68, 85],
    'Age': [25, 30, 22, 35, 28, 24, 33, 26, 32, 40]
}

# Create a DataFrame
df = pd.DataFrame(data)

# Create a pair plot
sns.pairplot(df)

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



Explanation:

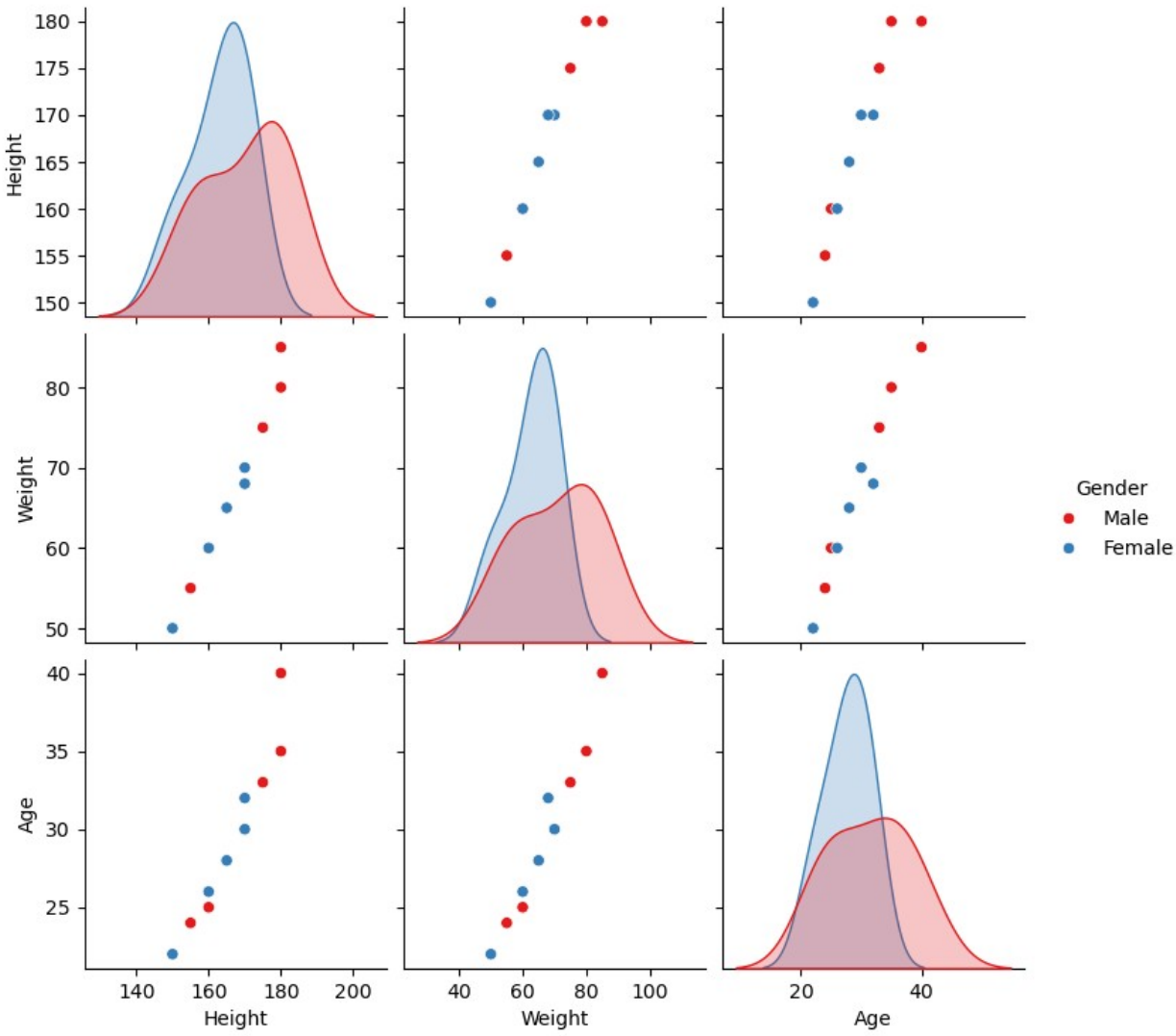
- **`sns.pairplot(df)`**: Creates a pair plot for all the features in the DataFrame `df`. The scatter plots show relationships between the features (e.g., **Height vs. Weight**), and the diagonal shows histograms for the distribution of each individual feature (e.g., the distribution of **Height**).

Output: This pair plot will create a grid of scatter plots that visualize the pairwise relationships between **Height**, **Weight**, and **Age**.

Step 2: Enhancing the Pair Plot

Let's enhance the pair plot by coloring the data points based on a **categorical feature** (for example, **gender**) and adding **KDE plots** on the diagonal.

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

# Sample data
data = {
    'Height': [160, 170, 150, 180, 165, 155, 175, 160, 170, 180],
    'Weight': [60, 70, 50, 80, 65, 55, 75, 60, 68, 85],
    'Age': [25, 30, 22, 35, 28, 24, 33, 26, 32, 40],
    'Gender': ['Male', 'Female', 'Female', 'Male', 'Female', 'Male',
'Male', 'Female', 'Female', 'Male']
}

# Create a DataFrame
df = pd.DataFrame(data)

# Create an enhanced pair plot with color based on gender
sns.pairplot(df, hue="Gender", diag_kind="kde", palette="Set1")

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



Enhancements Explained:

- **`hue="Gender"`**: Colors the data points based on the **Gender** feature. This allows you to see how the relationships differ between males and females.
- **`diag_kind="kde"`**: Replaces the histogram on the diagonal with a **KDE (Kernel Density Estimate)** plot, which smooths out the distribution to give a better sense of the data's shape.
- **`palette="Set1"`**: Sets the color palette for the different genders.

Output: The enhanced pair plot will use different colors to represent **Male** and **Female**, allowing for a comparison of the relationships between features by gender.

Why Use Pair Plots?

1. **Explore Relationships Between Variables**:

- Pair plots show **pairwise relationships** between variables, which helps you understand how features are related to each other. For instance, if there's a **strong positive correlation** between two variables, it will be visible in the scatter plot.

2. **Visualize Distributions**:
   - The diagonal histograms or KDE plots show the distribution of individual features, helping you understand the **spread**, **central tendency**, and **skewness** of each variable.
3. **Compare Categorical Variables**:
   - Pair plots allow you to color-code based on a categorical variable (e.g., gender, class labels), making it easier to see how different groups behave in relation to each other.

Key Features of a Pair Plot:

1. **Scatter Plots**:
   - These plots show the **pairwise relationships** between numerical variables. For example, the plot between **Height** and **Weight** shows how these two features are related (whether they increase or decrease together).
2. **Diagonal Plots**:
   - On the diagonal of the grid, you can see the **distribution** of each individual feature (either as a **histogram** or a **KDE plot**).
3. **Color-Coding with Categorical Variables**:
   - Pair plots allow you to **color-code** the scatter plots based on a categorical feature (e.g., gender, target class), which helps compare distributions or relationships between different groups.
4. **Customizable Diagonal**:
   - You can use **histograms** or **KDE plots** to visualize the distribution of each feature on the diagonal.

Real-World Use Case for Pair Plots:

Pair plots are often used in **exploratory data analysis (EDA)** to inspect the relationships between features. For example, if you're working on a **real estate** dataset, you can use a pair plot to visualize how different variables like **square footage**, **number of bedrooms**, and **price** are related. The pair plot will help you spot trends, correlations, and potential outliers in the data.

Common Customizations for Pair Plots:

1. **Color Coding (`hue`)**: Use a categorical variable to color-code the data points.
2. **Diagonal Plot Type (`diag_kind`)**: Choose between a histogram or KDE plot for visualizing the distribution of individual features.
3. **Marker Customization**: Customize the shape, size, and color of the markers for different groups.

4. **Plot Size (`height`)**: Adjust the size of the plots using the `height` parameter in `sns.pairplot()`.

```
```

Here are coding examples for the remaining plots. You can try plotting these.

---

10. Bubble Plot

A **bubble plot** is a variation of the scatter plot, where the size of the bubbles represents a third variable.

```
import matplotlib.pyplot as plt

# Sample data
x = [10, 20, 30, 40, 50]
y = [15, 25, 35, 45, 55]
bubble_size = [100, 200, 300, 400, 500]  # This controls the size of the bubbles

# Create the bubble plot
plt.scatter(x, y, s=bubble_size, alpha=0.5, c='green')

# Add labels and title
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.title('Bubble Plot Example')

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

---

11. Swarm Plot

A **swarm plot** shows the distribution of a categorical variable but avoids overlapping points by adjusting their positions.

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

# Sample data
tips = sns.load_dataset('tips')

# Create the swarm plot
plt.figure(figsize=(8, 6))
sns.swarmplot(x='day', y='total_bill', data=tips)
```

```
# Add labels and title
plt.xlabel('Day')
plt.ylabel('Total Bill')
plt.title('Swarm Plot Example')

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

---

12. Joint Plot

A **joint plot** combines a scatter plot and histograms (or KDE) for the two variables being compared.

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

# Sample data
tips = sns.load_dataset('tips')

# Create the joint plot
sns.jointplot(x='total_bill', y='tip', data=tips, kind='scatter')

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

---

13. FacetGrid

A **FacetGrid** helps visualize the distribution of one variable based on the levels of another categorical variable.

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

# Sample data
tips = sns.load_dataset('tips')

# Create the FacetGrid
g = sns.FacetGrid(tips, col='sex', hue='smoker')
g.map(plt.scatter, 'total_bill', 'tip').add_legend()

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

---

14. Treemap

A **treemap** is useful for visualizing hierarchical data, where the size and color of each rectangle represent two additional variables.

```
import matplotlib.pyplot as plt
import squarify  # You need to install the squarify library (pip
install squarify)

# Sample data
sizes = [500, 300, 200, 100]
labels = ['Category A', 'Category B', 'Category C', 'Category D']
colors = ['red', 'green', 'blue', 'orange']

# Create the treemap
plt.figure(figsize=(8, 6))
squarify.plot(sizes=sizes, label=labels, color=colors, alpha=0.7)

# Add title
plt.title('Treemap Example')

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

---

15. Sunburst Plot

A **sunburst plot** is a radial treemap that shows hierarchical data in layers.

```
import plotly.express as px

# Sample data
data = dict(
    character=["Earth", "Continent", "Continent", "Country",
"Country", "Country", "City", "City", "City"],
    parent=["", "Earth", "Earth", "Continent", "Continent",
"Continent", "Country", "Country", "Country"],
    value=[100, 50, 50, 20, 15, 15, 10, 5, 5]
)

# Create the sunburst plot
fig = px.sunburst(data, names='character', parents='parent',
values='value')

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

---

16. Radar Plot

A **radar plot** (also called a **spider plot**) is used to visualize multivariate data in the form of a web.

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

# Sample data
categories = ['Speed', 'Reliability', 'Comfort', 'Safety',
'Efficiency']
values = [4, 3, 4.5, 5, 4]

# Number of variables we're plotting
N = len(categories)

# Create the angle for each category
angles = np.linspace(0, 2 * np.pi, N, endpoint=False).tolist()

# The plot is a circle, so we need to "complete the loop"
values += values[:1]
angles += angles[:1]

# Create the radar chart
fig, ax = plt.subplots(figsize=(6, 6), subplot_kw=dict(polar=True))
ax.fill(angles, values, color='green', alpha=0.25)
ax.plot(angles, values, color='green', linewidth=2)

# Add the categories
ax.set_xticks(angles[:-1])
ax.set_xticklabels(categories)

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

---

17. 3D Plot

A **3D plot** helps in visualizing relationships between three variables.

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

# Sample 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 the 3D plot
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.plot_surface(x, y, z, cmap='viridis')

# Add labels and title
ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_zlabel('Z')
plt.title('3D Plot Example')

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

---

18. Word Cloud

A **word cloud** displays text data where the size of each word represents its frequency.

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

# Sample text data
text = "Data Science Machine Learning Python AI Data Analysis
Visualization"

# Create the word cloud
wordcloud = WordCloud(background_color='white').generate(text)

# Display the word cloud
plt.figure(figsize=(8, 6))
plt.imshow(wordcloud, interpolation='bilinear')
plt.axis('off')  # No axis lines
plt.title('Word Cloud Example')
plt.show()
```

---

19. Waterfall Chart

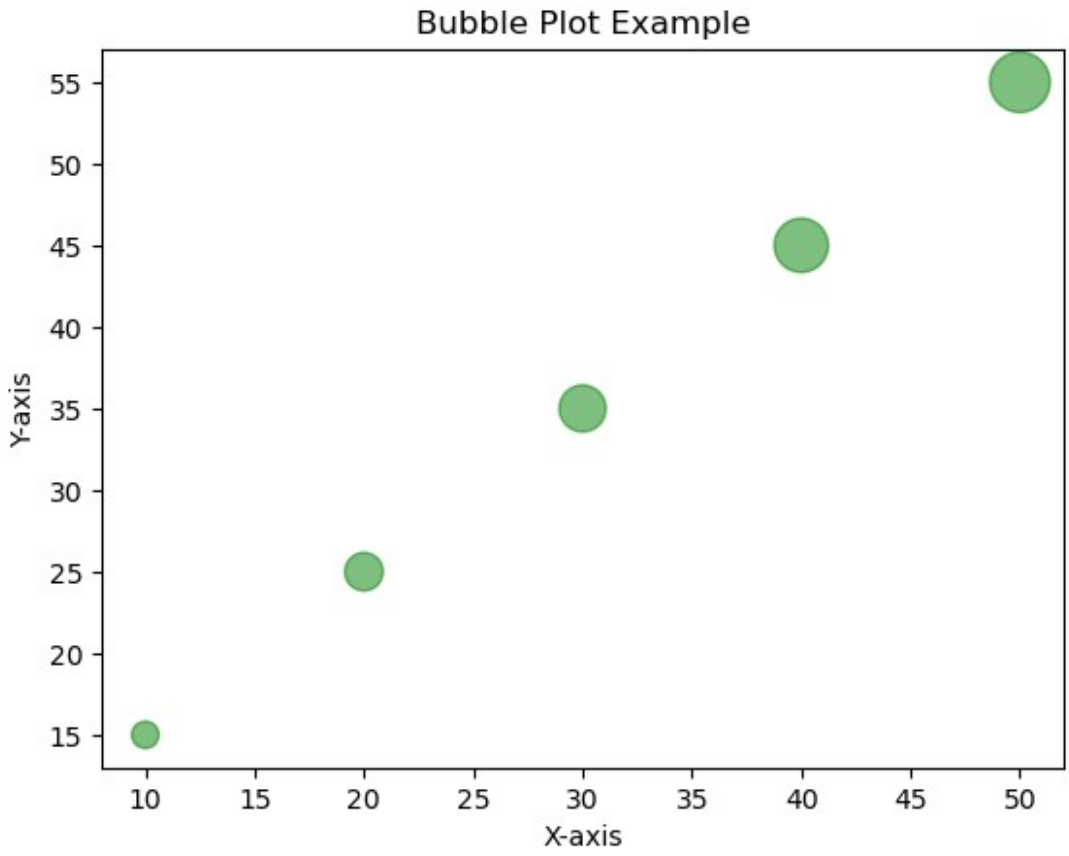
A **waterfall chart** shows the cumulative effect of sequentially introduced positive or negative values.

```
import matplotlib.pyplot as plt
import waterfall_chart

# Sample data
categories = ['Start', 'Revenue', 'Costs', 'Profit']
```

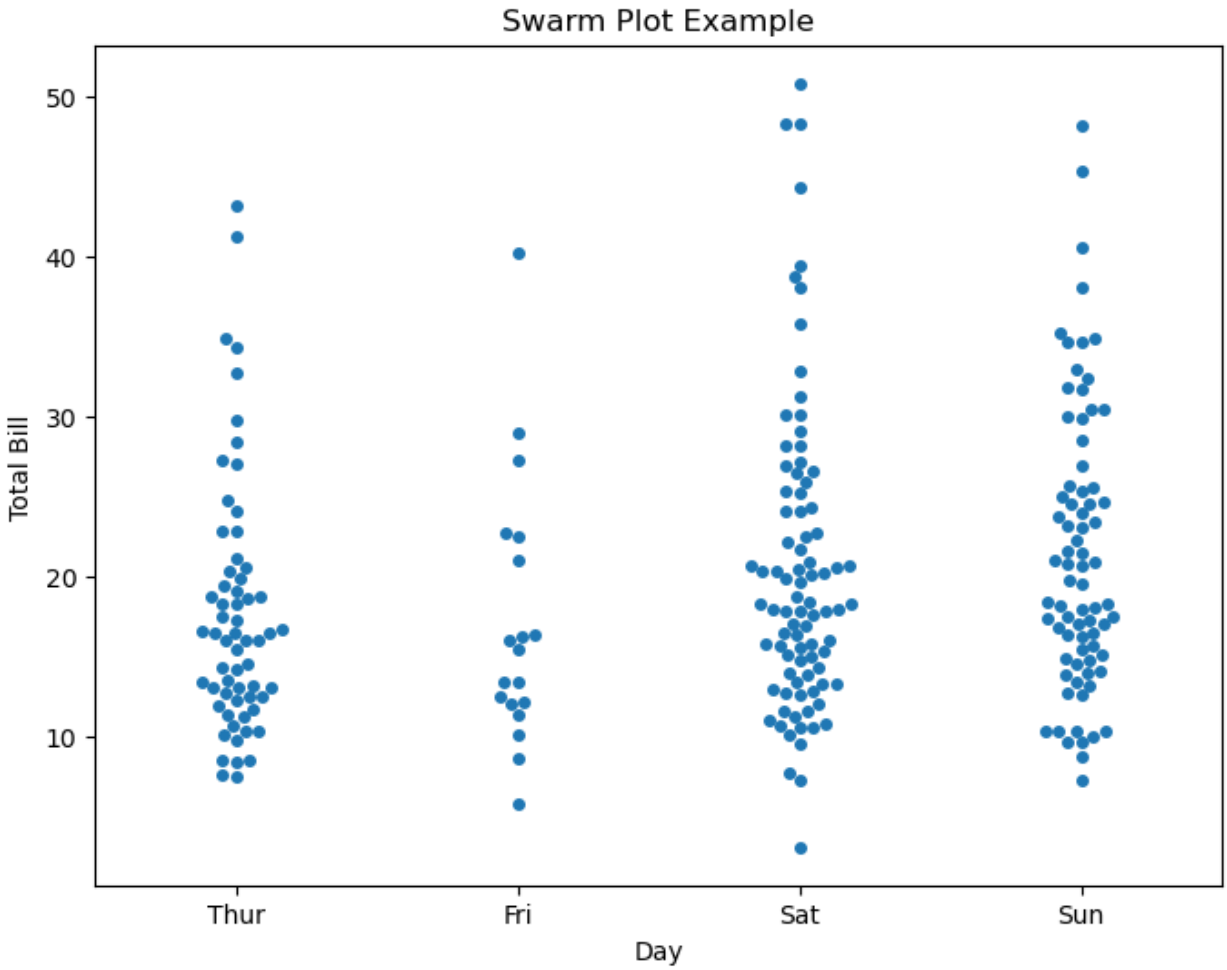
```
values = [500, 300, -200, 100]

# Create the waterfall chart
plt.figure(figsize=(8, 6))
waterfall_chart.plot(categories, values)

# Add title
plt.title('Waterfall Chart Example')

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

These examples should cover all the remaining types of plots! Feel free to run them and let me know if you have any questions or need help with specific customizations.

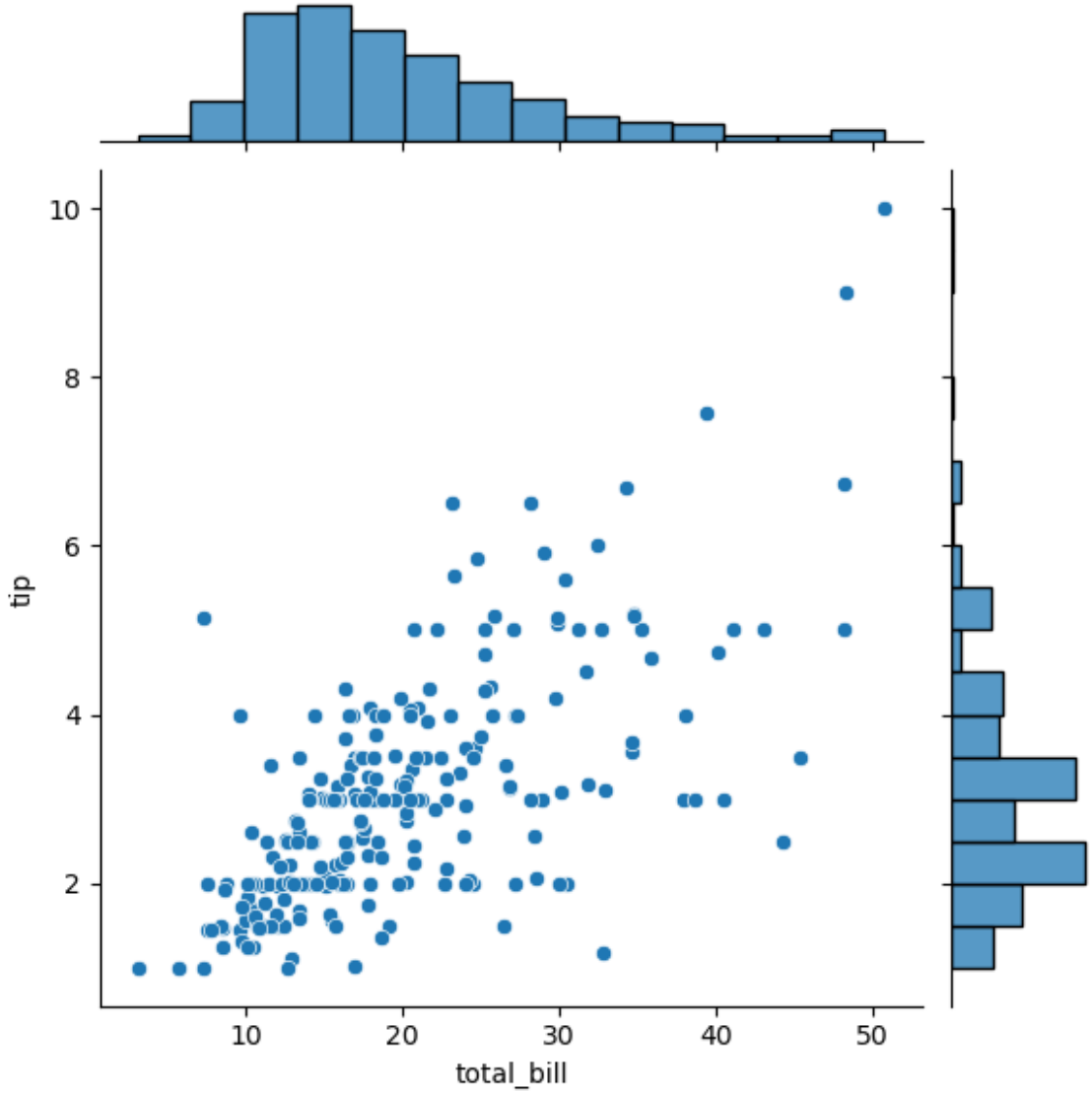
```
import matplotlib.pyplot as plt

# Sample data
x = [10, 20, 30, 40, 50]
y = [15, 25, 35, 45, 55]
bubble_size = [100, 200, 300, 400, 500]  # This controls the size of the bubbles

# Create the bubble plot
plt.scatter(x, y, s=bubble_size, alpha=0.5, c='green')

# Add labels and title
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.title('Bubble Plot Example')

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



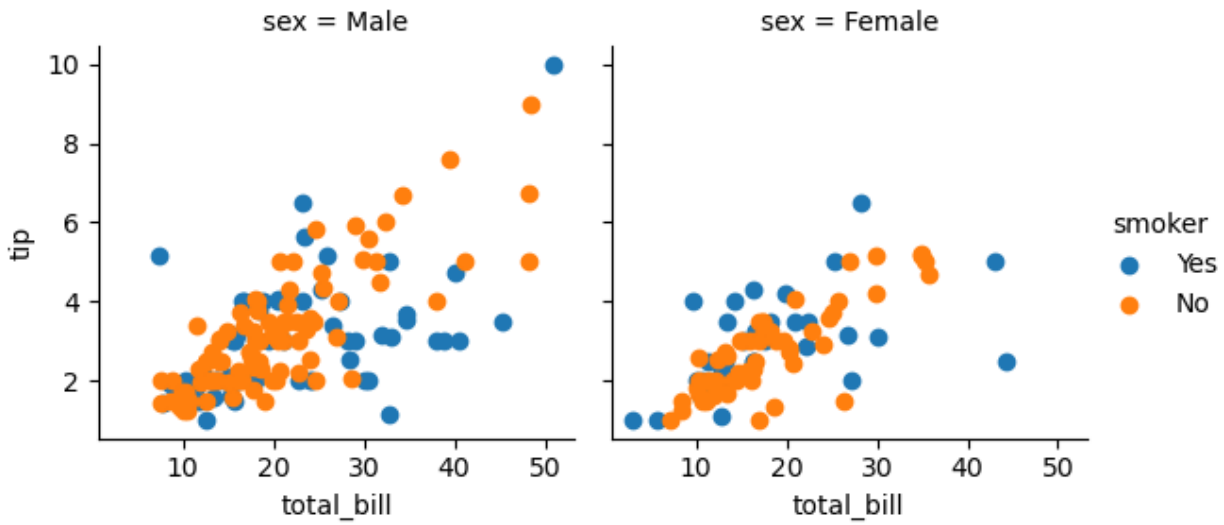

```
import seaborn as sns
import matplotlib.pyplot as plt

# Sample data
tips = sns.load_dataset('tips')

# Create the swarm plot
plt.figure(figsize=(8, 6))
sns.swarmplot(x='day', y='total_bill', data=tips)

# Add labels and title
plt.xlabel('Day')
plt.ylabel('Total Bill')
plt.title('Swarm Plot Example')

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



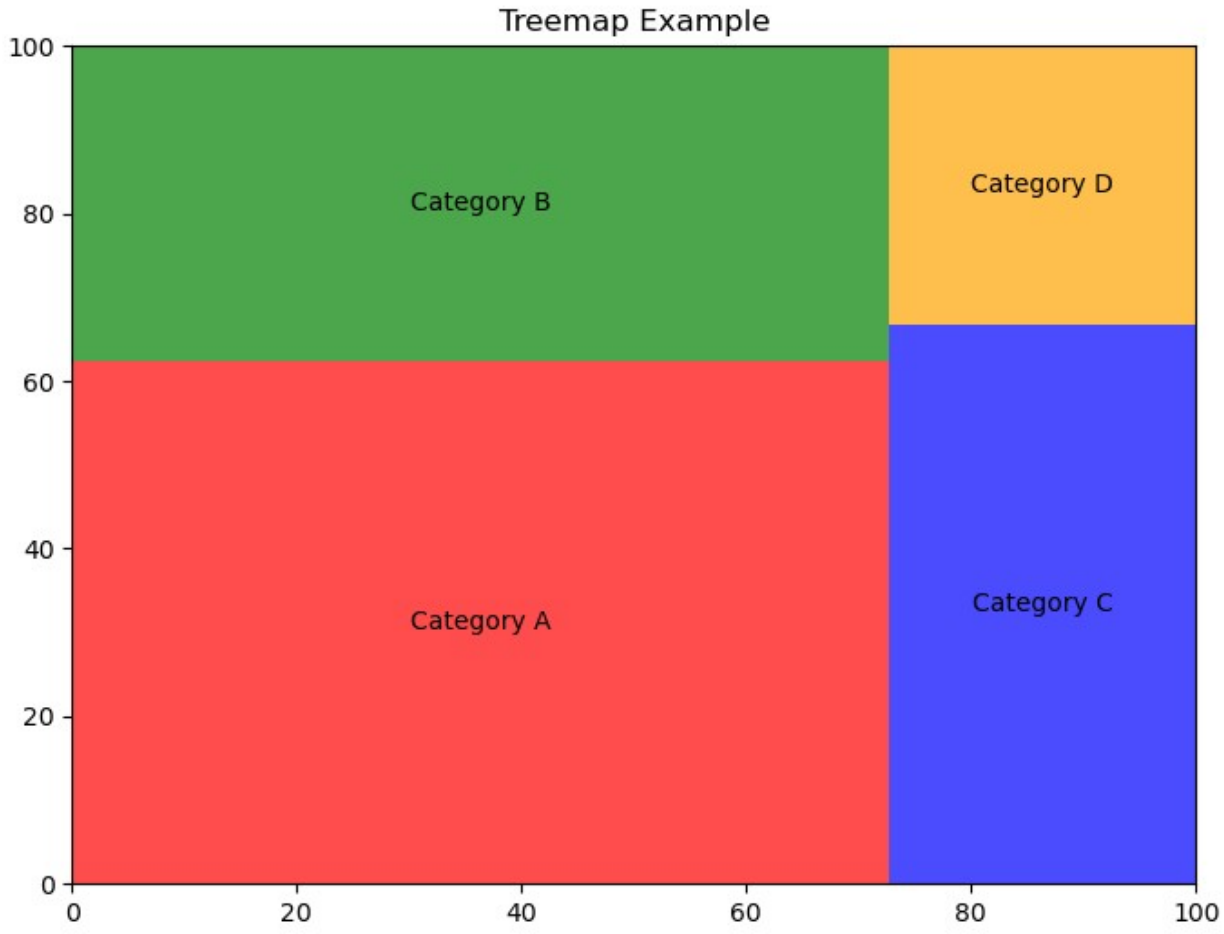

```
import seaborn as sns
import matplotlib.pyplot as plt

# Sample data
tips = sns.load_dataset('tips')

# Create the joint plot
sns.jointplot(x='total_bill', y='tip', data=tips, kind='scatter')

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




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

# Sample data
tips = sns.load_dataset('tips')

# Create the FacetGrid
g = sns.FacetGrid(tips, col='sex', hue='smoker')
g.map(plt.scatter, 'total_bill', 'tip').add_legend()

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

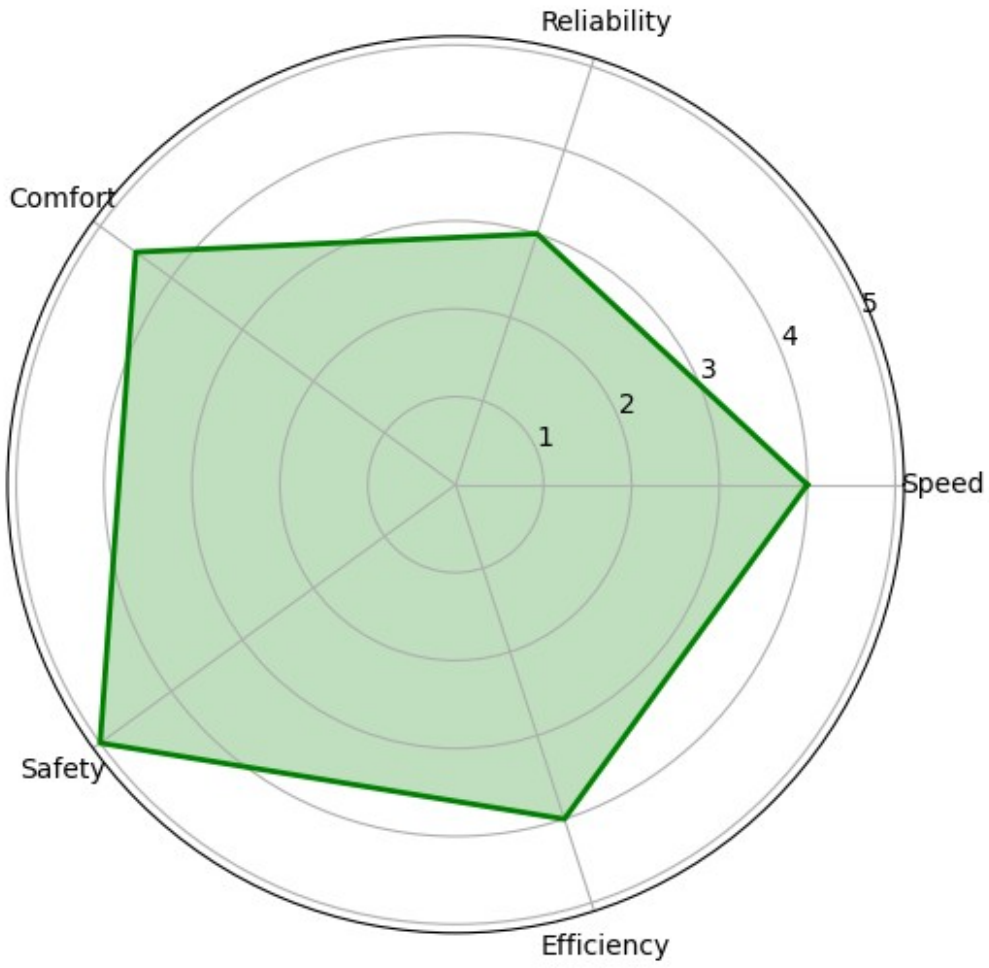



```
import matplotlib.pyplot as plt
import squarify  # You need to install the squarify library (pip
install squarify)

# Sample data
sizes = [500, 300, 200, 100]
labels = ['Category A', 'Category B', 'Category C', 'Category D']
colors = ['red', 'green', 'blue', 'orange']

# Create the treemap
plt.figure(figsize=(8, 6))
squarify.plot(sizes=sizes, label=labels, color=colors, alpha=0.7)

# Add title
plt.title('Treemap Example')

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



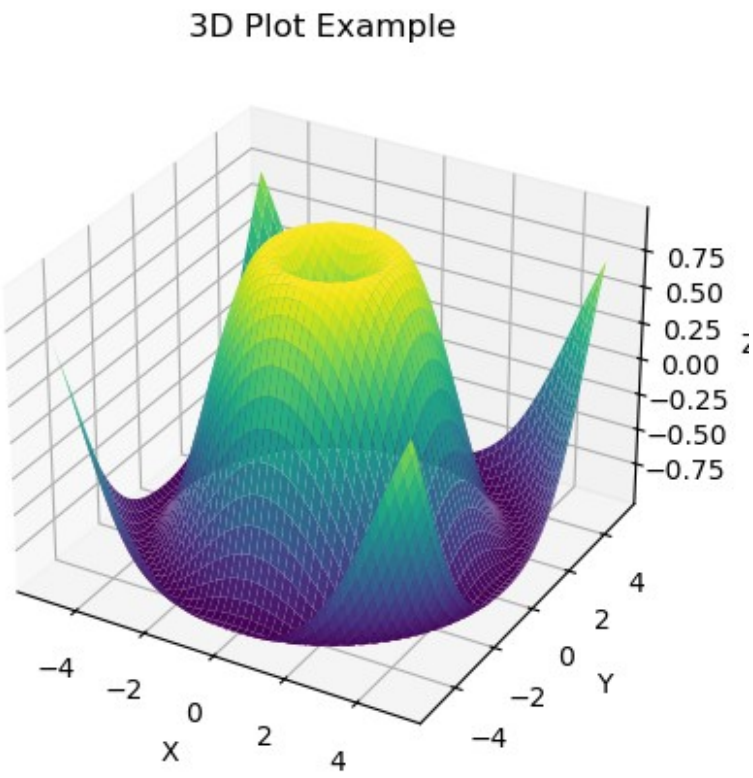

```
import matplotlib.pyplot as plt
import numpy as np

# Sample data
categories = ['Speed', 'Reliability', 'Comfort', 'Safety',
'Efficiency']
values = [4, 3, 4.5, 5, 4]

# Number of variables we're plotting
N = len(categories)

# Create the angle for each category
angles = np.linspace(0, 2 * np.pi, N, endpoint=False).tolist()

# The plot is a circle, so we need to "complete the loop"
values += values[:1]
angles += angles[:1]

# Create the radar chart
fig, ax = plt.subplots(figsize=(6, 6), subplot_kw=dict(polar=True))
ax.fill(angles, values, color='green', alpha=0.25)
```

```
ax.plot(angles, values, color='green', linewidth=2)

# Add the categories
ax.set_xticks(angles[:-1])
ax.set_xticklabels(categories)

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




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

# Sample 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 the 3D plot
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.plot_surface(x, y, z, cmap='viridis')

# Add labels and title
ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_zlabel('Z')
plt.title('3D Plot Example')

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




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

# Sample text data
text = "Data Science Machine Learning Python AI Data Analysis
Visualization"

# Create the word cloud
wordcloud = WordCloud(background_color='white').generate(text)

# Display the word cloud
```

```
plt.figure(figsize=(8, 6))
plt.imshow(wordcloud, interpolation='bilinear')
plt.axis('off')  # No axis lines
plt.title('Word Cloud Example')
plt.show()
```