VISUALIZATION

Visualization is a powerful technique for presenting information in a clear, engaging, and easily digestible way. By transforming data into visual representations, we can uncover patterns, trends, and insights that might otherwise be hidden or difficult to interpret.

Data visualization involves the use of graphical elements like charts, graphs, and maps to illustrate data relationships and patterns. This approach can enhance understanding, facilitate decision-making, and communicate complex information effectively

Python offers a rich ecosystem of libraries for data visualization, making it a versatile tool for analysts and data scientists. Some of the most popular libraries include:

- **Matplotlib:** A fundamental plotting library, providing a wide range of chart types and customization options.
- **Seaborn:** Built on top of Matplotlib, Seaborn offers a higher-level interface for creating aesthetically pleasing and informative visualizations.

MATPLOTLIB

Matplotlib is a powerful and versatile Python library for creating a wide range of static, animated, and interactive visualizations. Developed by John Hunter in 2002, it's built on top of NumPy, a fundamental library for numerical computations in Python.

Key components of a Matplotlib plot:

- **Figure:** The outermost container that holds all the plot elements.
- **Axes:** A region within the figure where data is plotted.
- **Axis:** The x-axis and y-axis that define the coordinate system within the axes.
- **Artists:** The individual graphical elements that make up the plot, such as lines, markers, text, and images.

Matplotlib's versatility and capabilities:

- **Static plots:** Create various chart types, including line plots, scatter plots, bar charts, histograms, and more.
- **Animated plots:** Generate dynamic visualizations to show changes in data over time.
- **Interactive plots:** Enable user interaction with plots through features like zooming, panning, and tooltips.
- **Customization:** Offer extensive customization options to tailor plots to specific needs and preferences.
- **Integration with other libraries:** Seamlessly integrate with other Python libraries like Pandas, Seaborn, and Plotly for advanced data analysis and visualization.

CUSTOMIZING VISUAL ELEMENTS FOR DETAILED PRESENTATIONS

Figure:

- The outermost container that holds all the plot elements, including axes, artists, and text.
- Can contain multiple plots or subplots arranged in a grid-like structure.
- Can be customized with properties like size, title, and background color.

Axes:

- A region within the figure where data is plotted.
- Defines the coordinate system for the plot.
- Can be customized with properties like labels, ticks, limits, and gridlines.

- Typically has two axes (x-axis and y-axis), but can also have a third axis (z-axis) for 3D plots.

Axis:

- The x-axis and y-axis that define the coordinate system within the axes.
- Responsible for generating ticks, labels, and limits.
- Can be customized with properties like tick labels, tick positions, and axis scaling.

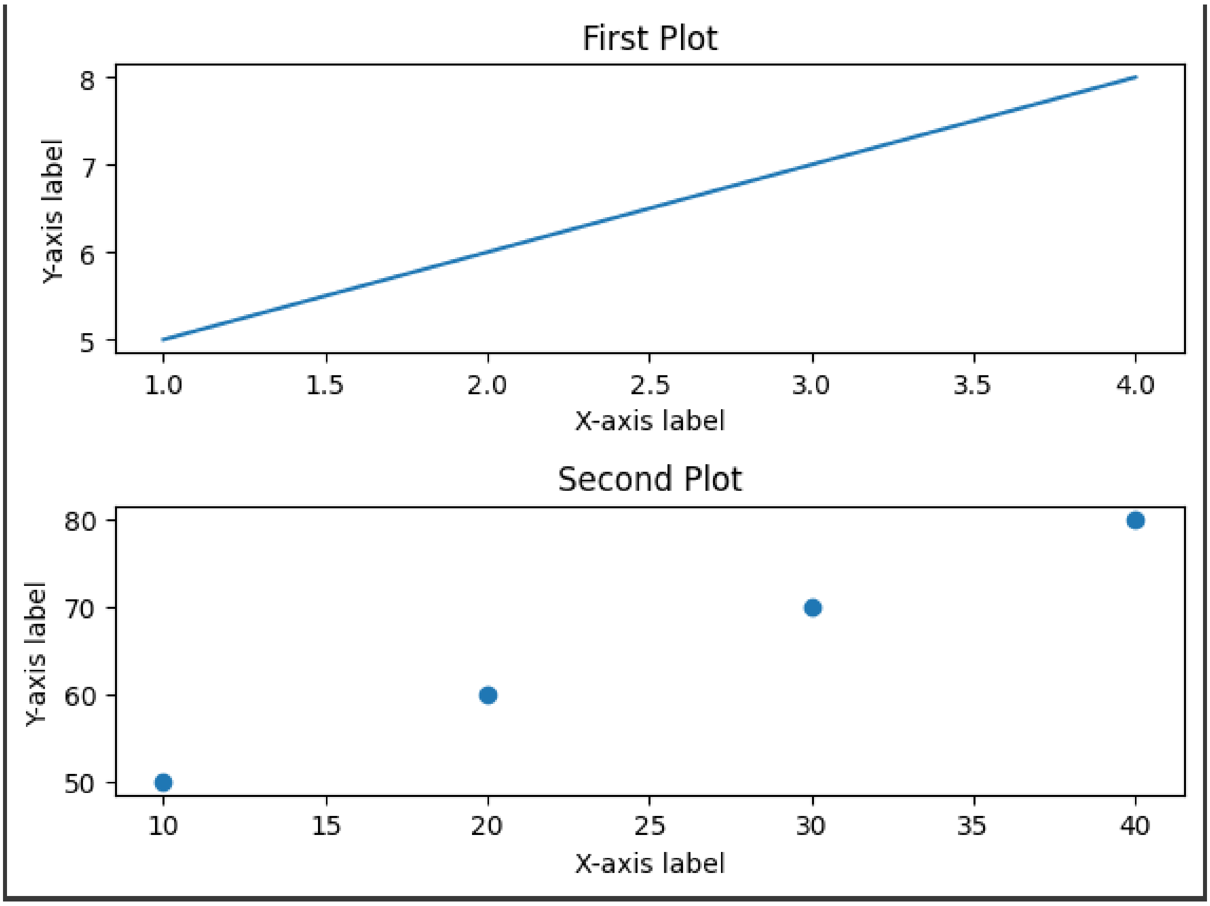
Artists:

- The individual graphical elements that make up the plot, such as lines, markers, text, images, and patches.
- Can be customized with properties like color, style, size, and position.

Example:

```
import matplotlib.pyplot as plt

# Create a figure with two subplots
fig, axs = plt.subplots(2, 1)

# Plot data on the first subplot
axs[0].plot([1, 2, 3, 4], [5, 6, 7, 8])
axs[0].set_xlabel('X-axis label')
axs[0].set_ylabel('Y-axis label')
axs[0].set_title('First Plot')

# Plot data on the second subplot
axs[1].scatter([10, 20, 30, 40], [50, 60, 70, 80])
axs[1].set_xlabel('X-axis label')
axs[1].set_ylabel('Y-axis label')
axs[1].set_title('Second Plot')

# Adjust spacing between subplots
plt.tight_layout()

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

OUTPUT:



PYPLOT

Pyplot is a submodule of Matplotlib that provides a convenient interface for creating plots. It offers a variety of plot types, including:

- **Bar graphs:** Visualize categorical data using rectangular bars.
- **Scatter plots:** Display the relationship between two numerical variables using points.
- **Pie charts:** Represent proportions of a whole using slices of a pie.
- **Histograms:** Show the distribution of a numerical variable using bins.
- **Area charts:** Plot the cumulative sum of a series of data points.

We import pyplot from matplotlib and numpy as:

```
[1]  import matplotlib.pyplot as plt
     import numpy as np
```

1.BAR GRAPH:

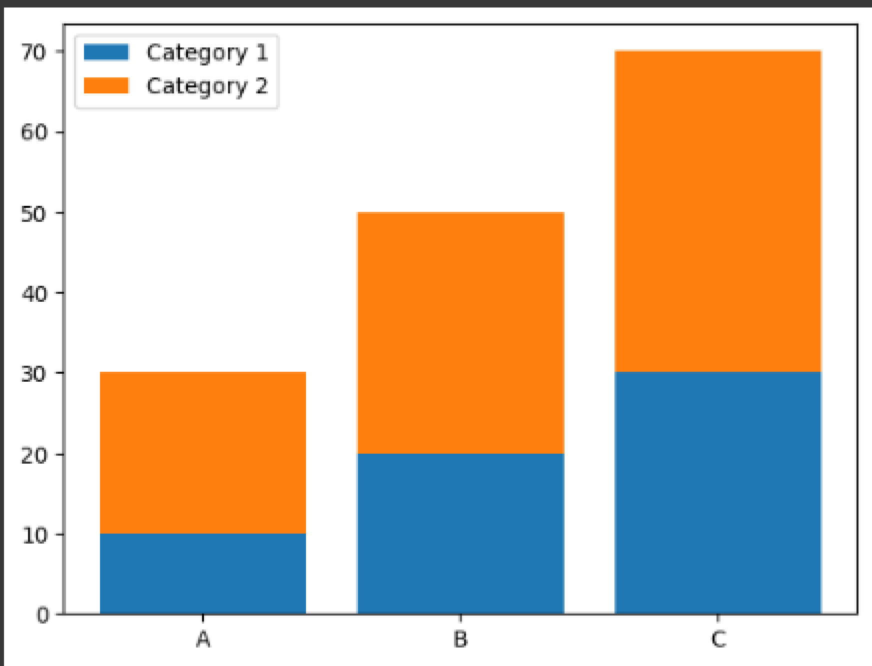
Bar Graphs

Enhanced Features:

- **Stacked Bar Graphs:** Visualize multiple categories within a single bar to compare their contributions.
- **Horizontal Bar Graphs:** Orient bars horizontally for better readability in specific scenarios.
- **Error Bars:** Indicate uncertainty or variability in data using error bars.
- **Grouping:** Group bars based on categories or treatments for easier comparison.

Example:

```python
import matplotlib.pyplot as plt

# Stacked bar graph
x = ['A', 'B', 'C']
y1 = [10, 20, 30]
y2 = [20, 30, 40]
plt.bar(x, y1, label='Category 1')
plt.bar(x, y2, label='Category 2', bottom=y1)
plt.legend()
plt.show()
```



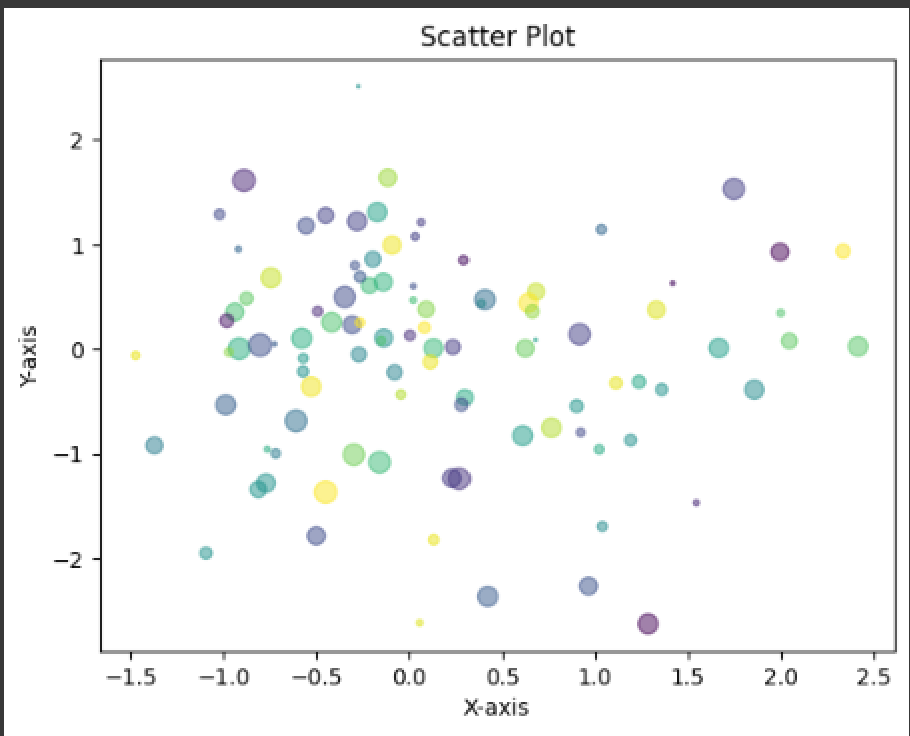
2. SCATTER PLOT:

Enhanced Features:

- **Color Coding:** Use different colors to represent different categories or groups within the data.
- **Size Variation:** Vary the size of data points based on a third variable to add another dimension.
- **Transparency:** Adjust the transparency of data points to handle overlapping points.
- **Annotation:** Add labels or annotations to specific data points for highlighting key information.

Example:

```
import matplotlib.pyplot as plt

x = np.random.randn(100)
y = np.random.randn(100)
sizes = 100 * np.random.rand(100)
colors = np.random.rand(100)

plt.scatter(x, y, c=colors, s=sizes, alpha=0.5)
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.title('Scatter Plot')
plt.show()
```



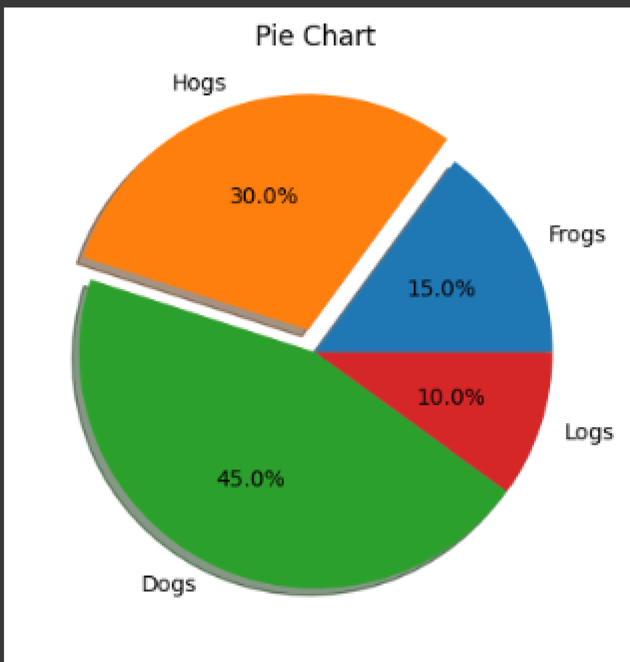
3. PIE CHART:

Enhanced Features:

- **Exploded Slices:** Emphasize specific slices by "exploding" them away from the pie.
- **Custom Labels:** Create custom labels for each slice, including percentages, values, or descriptions.
- **Shadow Effects:** Add depth and visual interest to the pie chart with shadows.
- **Autotext:** Automatically place labels within the slices with appropriate formatting.

Example:

```python
import matplotlib.pyplot as plt

labels = 'Frogs', 'Hogs', 'Dogs', 'Logs'
sizes = [15, 30, 45, 10]
explode = (0, 0.1, 0, 0)  # Only explode the second slice

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



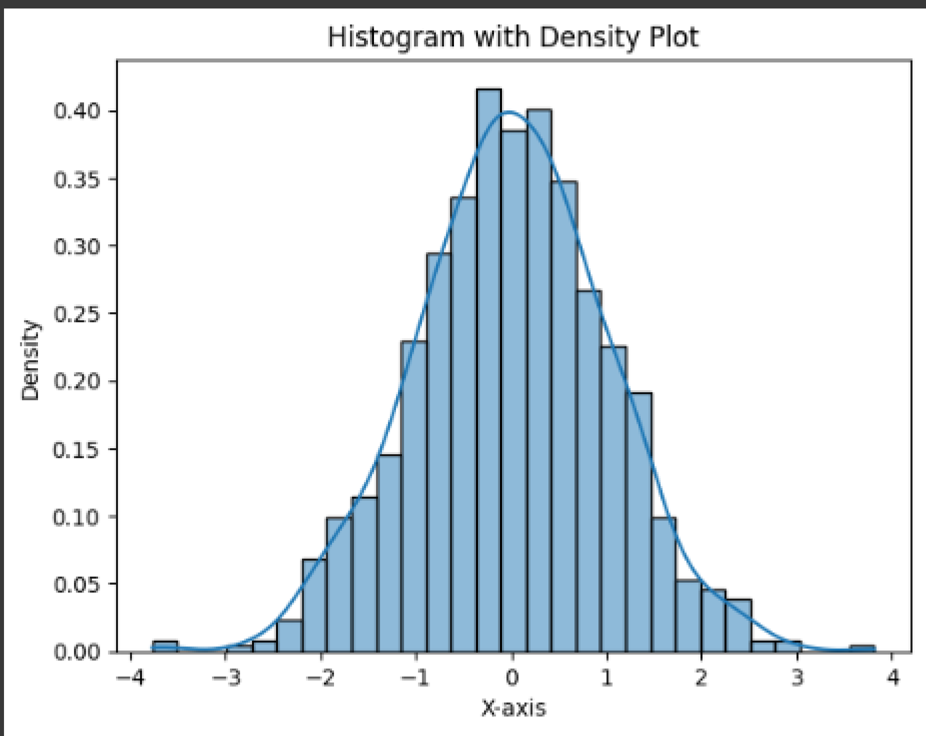
4. HISTOGRAM:

Enhanced Features:

- **Density Plots:** Overlay a density curve on the histogram to visualize the probability distribution.
- **Cumulative Histograms:** Show the cumulative distribution of the data.
- **Bin Customization:** Customize bin size, number of bins, and bin edges for better visualization.
- **Statistics:** Display statistical measures like mean, median, and mode on the histogram.

Example:

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

data = np.random.randn(1000)
sns.histplot(data, kde=True, stat='density')
plt.xlabel('X-axis')
plt.ylabel('Density')
plt.title('Histogram with Density Plot')
plt.show()
```



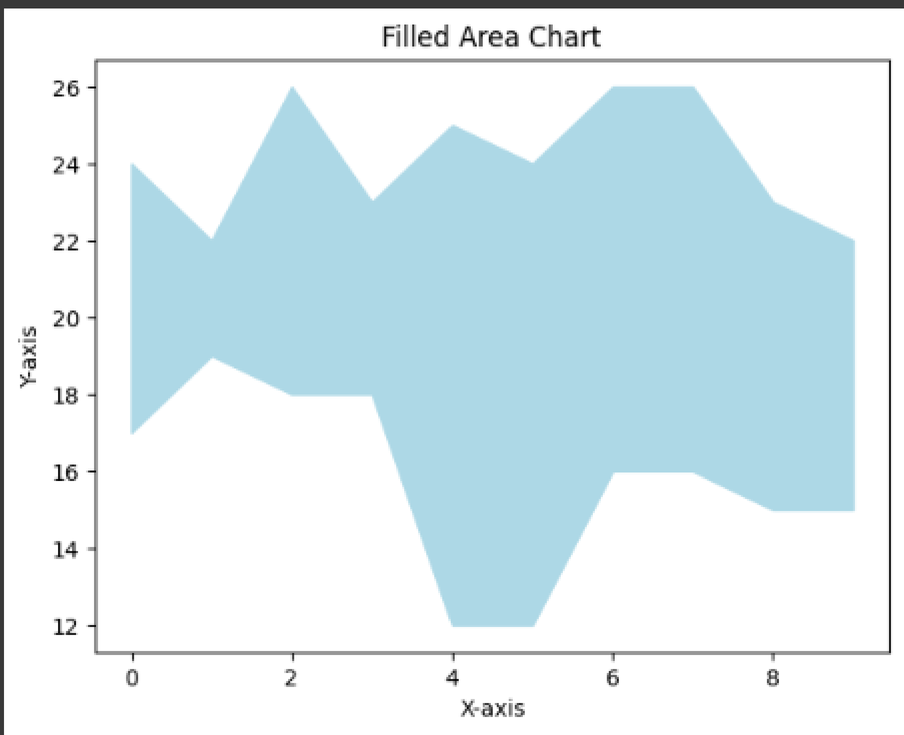
5. AREA CHART:

Enhanced Features:

- **Stacked Area Charts:** Visualize multiple series of data stacked on top of each other to show their cumulative contributions.
- **Filled Areas:** Fill areas between two lines or curves to highlight specific regions.
- **Alpha Blending:** Adjust the transparency of stacked areas to avoid obscuring details.
- **Gradient Fills:** Apply gradients to areas for more visual interest.

Example:

```python
import matplotlib.pyplot as plt

x = np.arange(10)
y1 = np.random.randint(10, 20, 10)
y2 = np.random.randint(20, 30, 10)

plt.fill_between(x, y1, y2, color='lightblue')
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.title('Filled Area Chart')
plt.show()
```



SEABORN

Seaborn is a powerful Python visualization library that builds upon Matplotlib, providing a higher-level interface for creating aesthetically pleasing and informative plots. It offers several advantages over Matplotlib, including:

- **Built-in defaults:** Seaborn comes with pre-defined styles and color palettes, making it easier to create visually appealing plots without extensive customization.
- **Advanced features:** It offers a range of advanced features, such as statistical plots, joint distributions, and categorical plots, that are not readily available in Matplotlib.
- **Seaborn-specific plot types:** Seaborn categorizes its plots into types such as relational, categorical, distribution, regression, and matrix plots, providing a structured approach to visualization.

Relational plots:

- **Scatter plots:** Show the relationship between two numerical variables.
- **Line plots:** Plot the values of a variable over time or another numerical variable.
- **Joint plots:** Combine a scatter plot with histograms of the marginal distributions.

Categorical plots:

- **Bar plots:** Visualize categorical data using rectangular bars.
- **Count plots:** Count the occurrences of values in a categorical variable.
- **Box plots:** Show the distribution of a numerical variable across different categories.

Distribution plots:

- **Histograms:** Show the distribution of a numerical variable using bins.
- **KDE plots:** Estimate the probability density function of a numerical variable.
- **Distplots:** Combine histograms and KDE plots.

Regression plots:

- **Regression lines:** Fit a linear regression model to the data and plot the line.
- **Residual plots:** Show the residuals of a regression model to assess the goodness of fit.

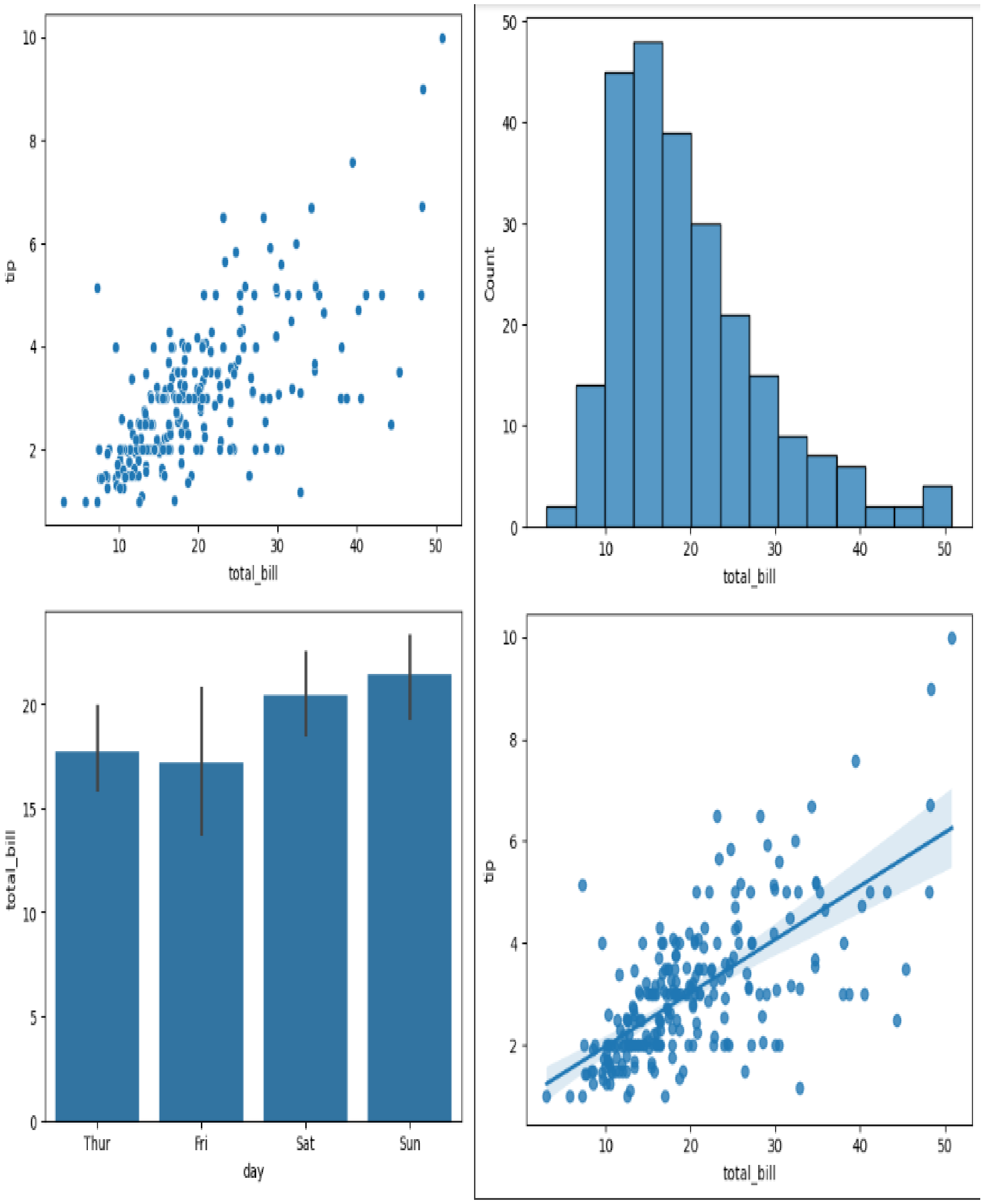
Matrix plots:

- **Heatmaps:** Visualize a matrix of values using a color scale.
- **Clustermaps:** Combine clustering and heatmaps to visualize patterns in data.

Example:

```
import seaborn as sns


# Load a dataset
tips = sns.load_dataset('tips')

# Create a scatter plot
sns.scatterplot(x='total_bill', y='tip', data=tips)
plt.show()

# Create a bar plot
sns.barplot(x='day', y='total_bill', data=tips)
plt.show()

# Create a histogram
sns.histplot(tips['total_bill'])
plt.show()

# Create a regression plot
sns.regplot(x='total_bill', y='tip', data=tips)
plt.show()
```

Output:


tip
total_bill
Count
total_bill
total_bill
day
Thur
Fri
Sat
Sun
tip
total_bill

It offers several unique features that make it a popular choice for data visualization:

- **Built-in themes:** Seaborn comes with pre-defined styles and color palettes, making it easy to create visually appealing plots without extensive customization.
- **Simplified syntax:** Seaborn provides a simplified syntax for creating complex visualizations, reducing the amount of code required compared to Matplotlib.
- **Integrated with Pandas data structures:** Seaborn is seamlessly integrated with Pandas, allowing you to work directly with DataFrames and Series for data visualization.
- **Exploring and understanding data:** Seaborn's functions are designed to help you explore and understand your data, making it easier to identify patterns and trends.
- **Visualizing statistical relationships:** Seaborn offers a variety of functions for visualizing statistical relationships, such as regression plots, joint plots, and pair plots.

PLOTS IN SEABORN

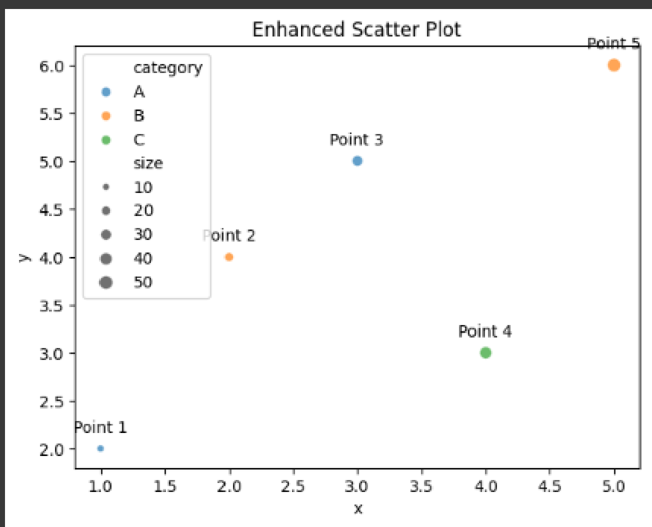
1. SCATTER PLOT:

Enhanced Features:

- **Color Coding: Use different colors to represent categories or groups within the data.**
- **Size Variation: Vary the size of data points based on a third variable.**
- **Transparency: Adjust the transparency of data points to handle overlapping points.**
- **Annotation: Add labels or annotations to specific data points for highlighting key information.**

Example

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

# Load sample data (replace with your own data)
data = {'x': [1, 2, 3, 4, 5],
        'y': [2, 4, 5, 3, 6],
        'category': ['A', 'B', 'A', 'C', 'B'],
        'size': [10, 20, 30, 40, 50],
        'annotations': ['Point 1', 'Point 2', 'Point 3', 'Point 4', 'Point 5']}
df = pd.DataFrame(data)

# Create the scatter plot with enhanced features
sns.scatterplot(x='x', y='y', data=df, hue='category', size='size', alpha=0.7)

# Add annotations to specific points
for i, row in df.iterrows():
    plt.annotate(row['annotations'], (row['x'], row['y']), textcoords='offset points', xytext=(0, 10), ha='center')

plt.title('Enhanced Scatter Plot')
plt.show()
```



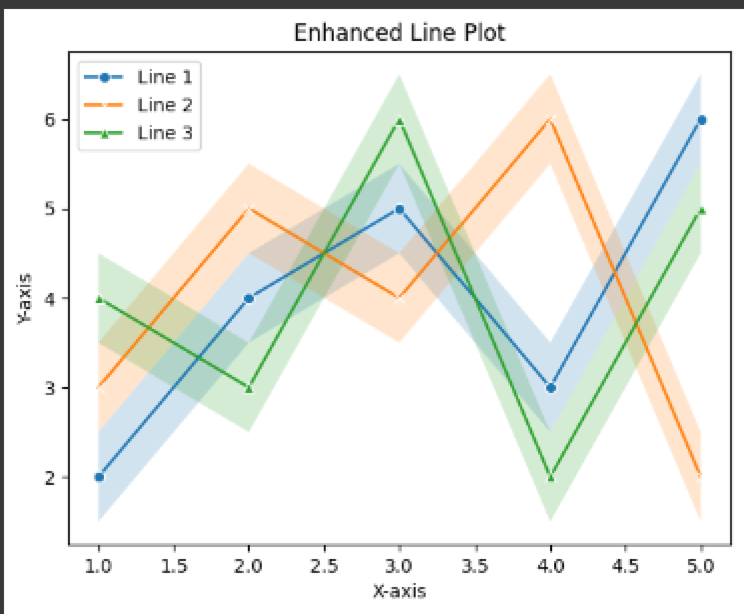
2. LINE PLOT

Enhanced Features:

- **Multiple Lines:** Plot multiple lines on the same axes for comparison.
- **Error Bands:** Add confidence intervals or error bands to indicate uncertainty.
- **Markers:** Use markers to highlight specific data points.
- **Interpolations:** Use interpolation methods to smooth the line between data points.

Example

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

# Sample data (replace with your own)
data = {'x': [1, 2, 3, 4, 5],
        'y1': [2, 4, 5, 3, 6],
        'y2': [3, 5, 4, 6, 2],
        'y3': [4, 3, 6, 2, 5]}
df = pd.DataFrame(data)

# Create a line plot with multiple lines, error bands, markers, and interpolation
sns.lineplot(x='x', y='y1', data=df, label='Line 1', marker='o')
sns.lineplot(x='x', y='y2', data=df, label='Line 2', marker='x')
sns.lineplot(x='x', y='y3', data=df, label='Line 3', marker='^')

# Add error bands (replace with actual error data)
plt.fill_between(df['x'], df['y1'] - 0.5, df['y1'] + 0.5, alpha=0.2)
plt.fill_between(df['x'], df['y2'] - 0.5, df['y2'] + 0.5, alpha=0.2)
plt.fill_between(df['x'], df['y3'] - 0.5, df['y3'] + 0.5, alpha=0.2)

plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.title('Enhanced Line Plot')
plt.legend()
plt.show()
```



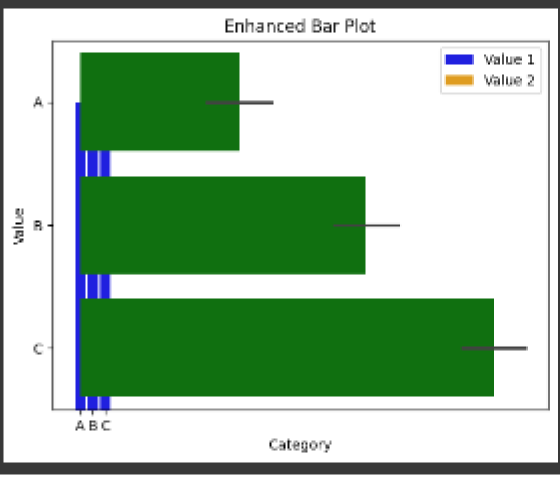
3. BAR PLOT:

Enhanced Features:

- **Stacked Bar Graphs: Visualize multiple categories within a single bar to compare their contributions.**
- **Horizontal Bar Graphs: Orient bars horizontally for better readability in specific scenarios.**
- **Error Bars: Indicate uncertainty or variability in data using error bars.**
- **Grouping: Group bars based on categories or treatments for easier comparison.**

Example:

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

# Sample data (replace with your own)
data = {'category': ['A', 'B', 'C', 'A', 'B', 'C'],
        'value1': [10, 20, 30, 15, 25, 35],
        'value2': [5, 10, 15, 8, 12, 18]}
df = pd.DataFrame(data)

# Create a stacked bar plot with error bars
# Calculate the bottom for each category by grouping and summing 'value1'
bottom_values = df.groupby('category')['value1'].sum().values

# Create the first bar plot for 'value1'
sns.barplot(x='category', y='value1', data=df, label='Value 1', color='blue')

# Create the second bar plot for 'value2' with the calculated bottom values
# Note: We use the index of the groupby result as the x-axis
sns.barplot(x=df.groupby('category')['value1'].sum().index,
            y='value2',
            data=df.groupby('category').sum().reset_index(),  # Aggregate data for value2
            label='Value 2',
            color='orange',
            bottom=bottom_values)

# Add error bars (replace with actual error data)
plt.errorbar(x=df['category'], y=df['value1'] + df['value2'], yerr=1, fmt='o')

# Create a horizontal bar plot
sns.barplot(x='value1', y='category', data=df, orient='h', color='green')

# Group bars based on a third category (replace with your own data)
# Assuming 'group' column exists in your DataFrame
# sns.barplot(x='category', y='value1', data=df, hue='group', palette='colorblind')

plt.xlabel('Category')
plt.ylabel('Value')
plt.title('Enhanced Bar Plot')
plt.legend()
plt.show()
```



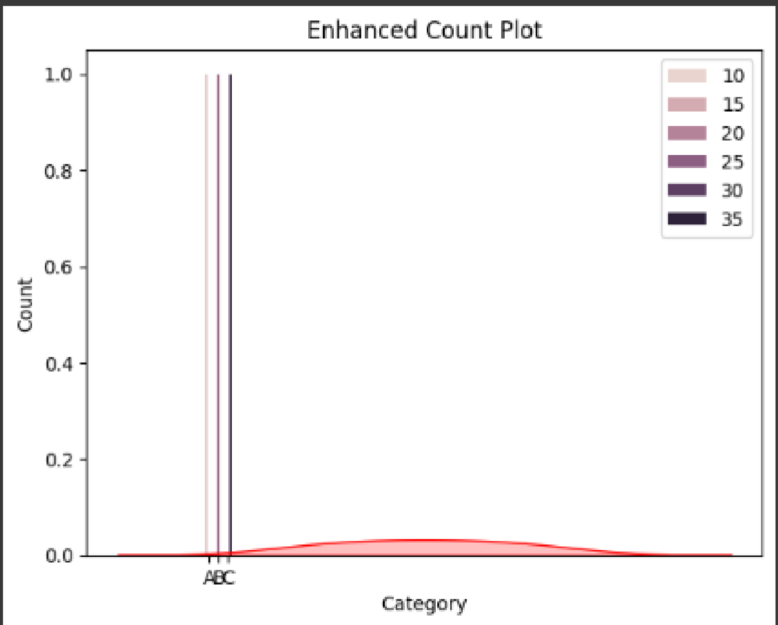
4. COUNT PLOT:

Enhanced Features:

- **Horizontal Orientation:** Plot the bars horizontally.
- **Hue:** Group the bars by a categorical variable.
- **Estimation Plot:** Overlay an estimate plot (e.g., KDE) on the count plot.

Example:

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

# Sample data (replace with your own)
data = {'category': ['A', 'B', 'C', 'A', 'B', 'C'],
        'value': [10, 20, 30, 15, 25, 35]}
df = pd.DataFrame(data)

# Create a count plot with horizontal orientation, grouping, and an estimation plot
sns.countplot(x='category', data=df, hue='value', orient='h')
sns.kdeplot(df['value'], color='red', fill=True)

plt.xlabel('Category')
plt.ylabel('Count')
plt.title('Enhanced Count Plot')
plt.legend()
plt.show()
```



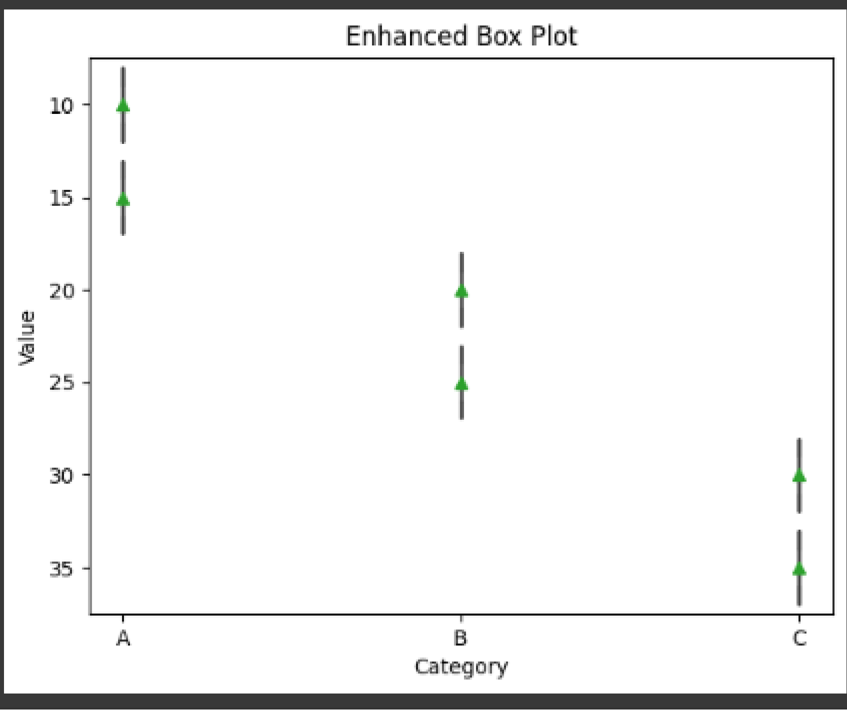
5. BOX PLOT:

Enhanced Features:

- **Horizontal Orientation:** Plot the boxes horizontally.
- **Hue:** Group the boxes by a categorical variable.
- **Notch:** Display notches on the boxes to indicate confidence intervals.
- **Showmeans:** Show the mean value within each box.

Example

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

# Sample data (replace with your own)
data = {'category': ['A', 'B', 'C', 'A', 'B', 'C'],
        'value': [10, 20, 30, 15, 25, 35]}
df = pd.DataFrame(data)

# Create a box plot with enhanced features
sns.boxplot(x='category', y='value', data=df, orient='h', notch=True, showmeans=True)

plt.xlabel('Category')
plt.ylabel('Value')
plt.title('Enhanced Box Plot')
plt.show()
```



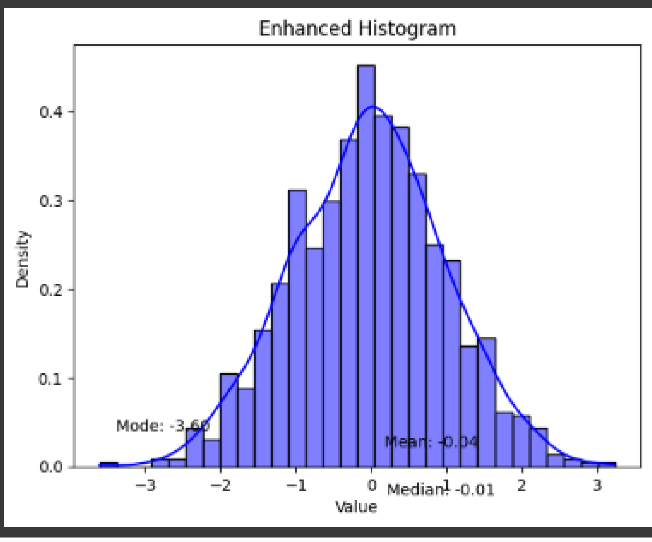
6. HISTOPLOT :

Enhanced Features:

- **Density Plots:** Overlay a density curve on the histogram to visualize the probability distribution.
- **Cumulative Histograms:** Show the cumulative distribution of the data.
- **Bin Customization:** Customize bin size, number of bins, and bin edges for better visualization.
- **Statistics:** Display statistical measures like mean, median, and mode on the histogram.

Example

```python
import seaborn as sns
import matplotlib.pyplot as plt
import numpy as np
from scipy import stats # Import the stats module from scipy
# Sample data (replace with your own)
data = np.random.randn(1000)
# Create a histplot with enhanced features
sns.histplot(data, kde=True, stat='density', cumulative=False, bins=30, color='blue')
# Add statistical annotations
mean = np.mean(data)
median = np.median(data)
# Calculate the mode using scipy.stats.mode, and handle the case where it returns a scalar
mode_result = stats.mode(data)
# Check if the mode is a scalar or an array
if isinstance(mode_result.mode, np.ndarray):
    mode = mode_result.mode[0]  # Access the first element if it's an array
else:
    mode = mode_result.mode  # Use the scalar value directly if it's not an array

plt.annotate(f'Mean: {mean:.2f}', xy=(mean, 0), xytext=(10, 10), textcoords='offset points', ha='left', va='bottom')
plt.annotate(f'Median: {median:.2f}', xy=(median, 0), xytext=(10, -10), textcoords='offset points', ha='left', va='top')
plt.annotate(f'Mode: {mode:.2f}', xy=(mode, 0), xytext=(10, 20), textcoords='offset points', ha='left', va='bottom')

plt.xlabel('Value')
plt.ylabel('Density')
plt.title('Enhanced Histogram')
plt.show()
```



7. KDE PLOT:

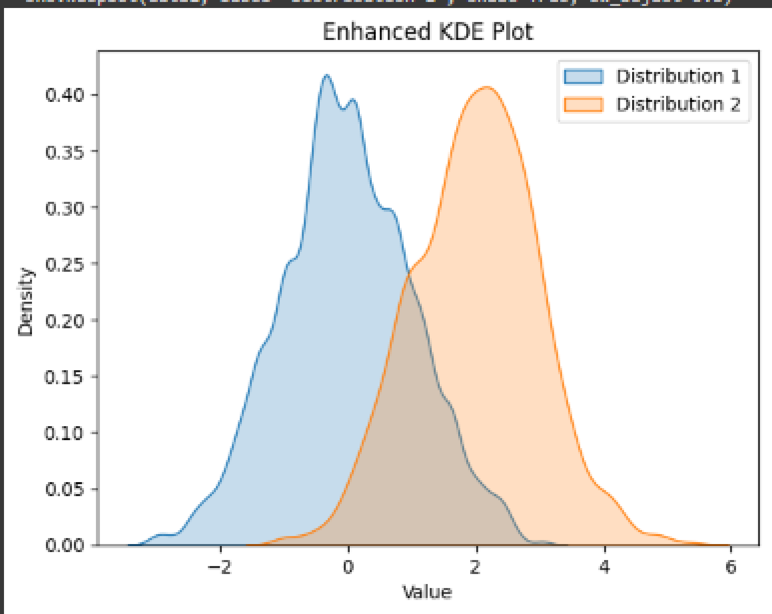
Enhanced Features:

- **Multiple KDEs:** Plot multiple KDEs on the same axes for comparison.
- **Shading:** Shade the area under the KDE curve.
- **Bandwidth:** Adjust the bandwidth of the KDE to control the smoothness.

Example

```python
# Sample data (replace with your own)
data1 = np.random.randn(1000)
data2 = np.random.normal(loc=2, scale=1, size=1000)

# Create a KDE plot with multiple KDEs, shading, and bandwidth adjustment
sns.kdeplot(data1, label='Distribution 1', shade=True, bw_adjust=0.5)
sns.kdeplot(data2, label='Distribution 2', shade=True, bw_adjust=0.8)

plt.xlabel('Value')
plt.ylabel('Density')
plt.title('Enhanced KDE Plot')
plt.legend()
plt.show()
```

```
<ipython-input-30-8f30eb25e17f>:6: FutureWarning:

`shade` is now deprecated in favor of `fill`; setting `fill=True`.
This will become an error in seaborn v0.14.0; please update your code.

  sns.kdeplot(data1, label='Distribution 1', shade=True, bw_adjust=0.5)
<ipython-input-30-8f30eb25e17f>:7: FutureWarning:

`shade` is now deprecated in favor of `fill`; setting `fill=True`.
This will become an error in seaborn v0.14.0; please update your code.

  sns.kdeplot(data2, label='Distribution 2', shade=True, bw_adjust=0.8)
```



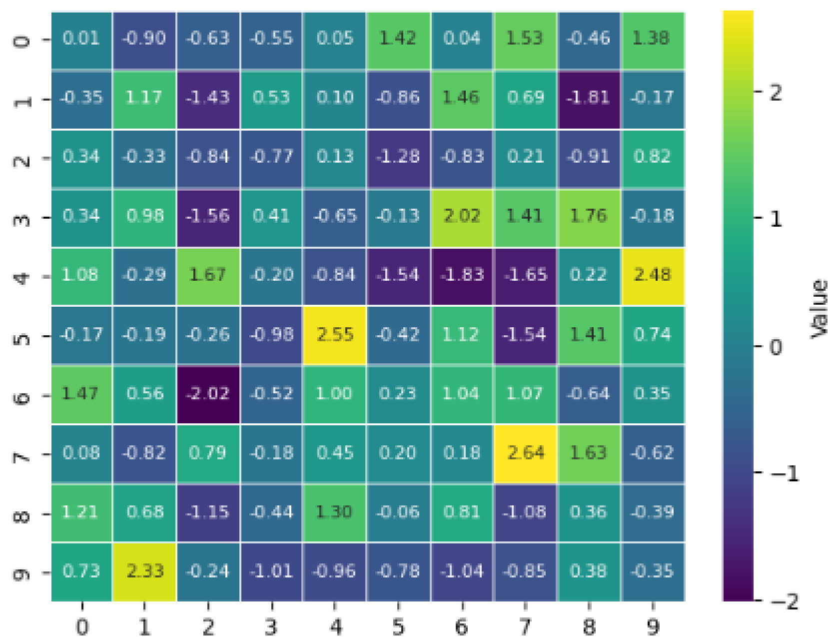
8. HEATMAP:

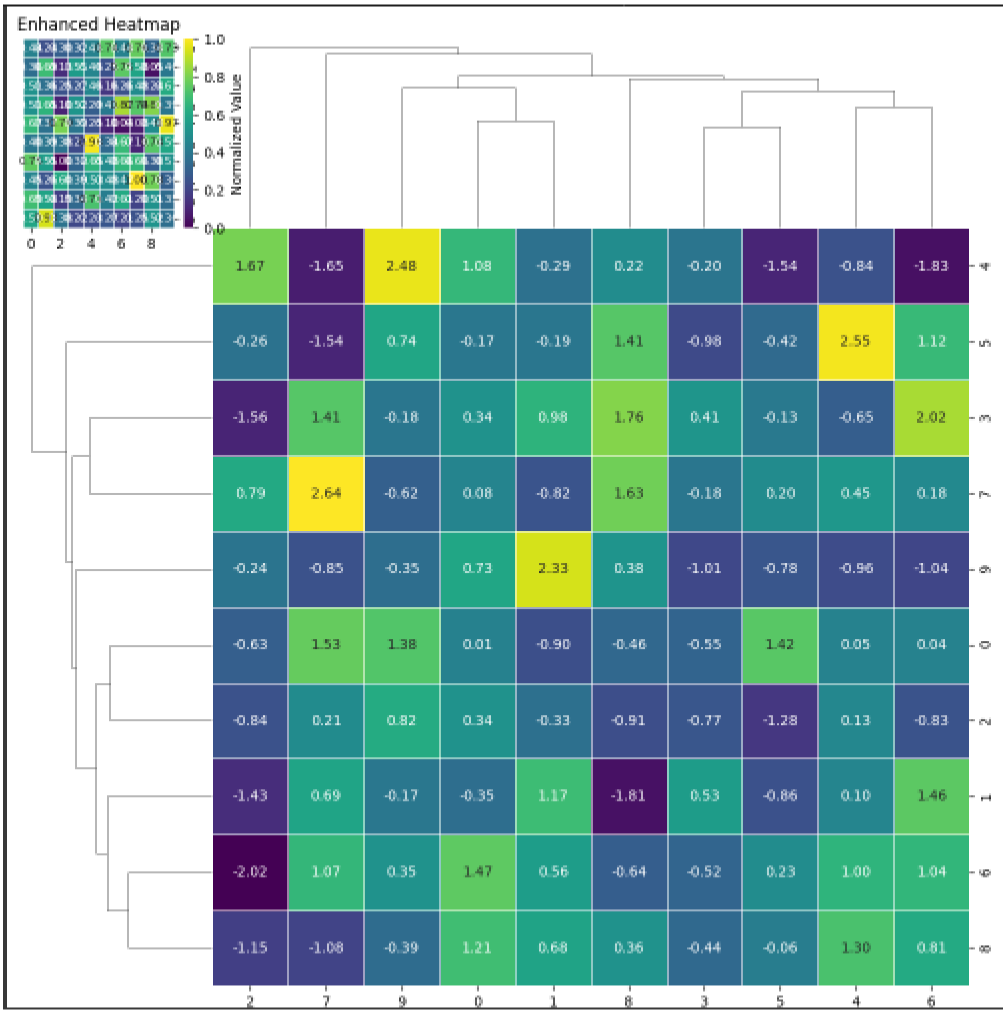
Enhanced Features:

- **Annotation: Add numerical values or annotations to each cell.**
- **Colormaps: Choose different colormaps to visualize the data in different ways.**
- **Clustering: Cluster the rows and columns to identify patterns.**
- **Normalization: Normalize the data to scale the values between 0 and 1**

Example

```python
import seaborn as sns
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np

# Sample data (replace with your own)
data = np.random.randn(10, 10)

# Create a heatmap with enhanced features
sns.heatmap(data, annot=True, fmt='.2f', cmap='viridis', linewidths=0.5, annot_kws={'size': 8}, cbar_kws={'label': 'Value'})

# Cluster rows and columns
sns.clustermap(data, annot=True, fmt='.2f', cmap='viridis', linewidths=0.5)

# Normalize data
data_normalized = (data - data.min()) / (data.max() - data.min())
sns.heatmap(data_normalized, annot=True, fmt='.2f', cmap='viridis', linewidths=0.5, annot_kws={'size': 8}, cbar_kws={'label': 'Normalized Value'})

plt.title('Enhanced Heatmap')
plt.show()
```





Enhanced Heatmap
Normalized Value

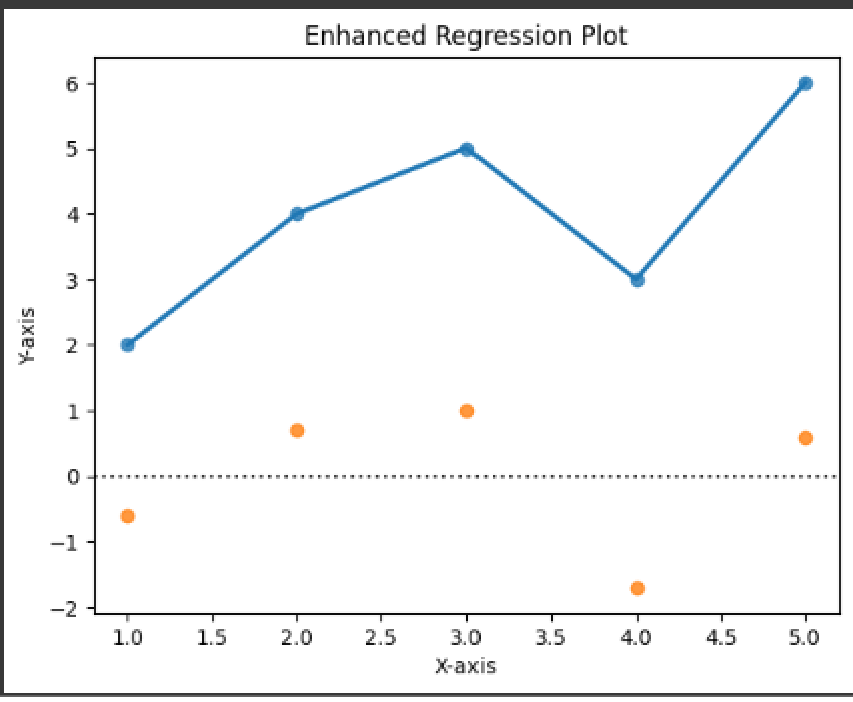
9. REG PLOT:

Enhanced Features:

- **Residual Plots: Add a residual plot to assess the goodness of fit.**
- **Confidence Intervals: Display confidence intervals for the regression line.**
- **Robust Regression: Use robust regression methods to handle outliers.**
- **Lowess Smoothing: Apply local weighted scatterplot smoothing to fit a non-parametric regression line.**

Example:

```python
import seaborn as sns
import matplotlib.pyplot as plt
import pandas as pd
import statsmodels.formula.api as smf
# Sample data (replace with your own)
data = {'x': [1, 2, 3, 4, 5],
        'y': [2, 4, 5, 3, 6]}
df = pd.DataFrame(data)
# Create a regplot with enhanced features
# Choose only ONE of the following regression methods:
# 1. LOWESS Regression:
sns.regplot(x='x', y='y', data=df, lowess=True, ci=95)
# 2. Polynomial Regression (order=1 is linear regression):
# sns.regplot(x='x', y='y', data=df, order=1, ci=95)
# 3. Robust Regression:
# sns.regplot(x='x', y='y', data=df, robust=True, ci=95)
# Add a residual plot
sns.residplot(x='x', y='y', data=df)
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.title('Enhanced Regression Plot')
plt.show()
```



COMPARISON OF MATPLOTLIB AND SEABORN

Matplotlib and Seaborn are both powerful Python libraries for data visualization, each with its own strengths and weaknesses. Here's a comparison of their key features:

MATPLOTLIB

- **Foundation:** Matplotlib is the fundamental plotting library in Python, providing a wide range of plot types and customization options.
- **Flexibility:** It offers fine-grained control over plot elements, allowing for highly customized visualizations.
- **Complexity:** Can be more complex to use, especially for beginners, due to its lower-level interface.
- **Performance:** Generally faster than Seaborn for simple plots due to its lower-level implementation.

Advantages of Matplotlib:

- **Flexibility:** Matplotlib offers extensive customization options, allowing you to create highly tailored visualizations to meet your specific needs.
- **Granular control:** You have fine-grained control over plot elements, enabling you to customize every aspect of your visualization.
- **Performance:** Matplotlib is generally faster than Seaborn for simple plots due to its lower-level implementation.
- **Wide range of plot types:** Matplotlib supports a vast array of plot types, from basic line plots to complex 3D visualizations.
- **Integration with other libraries:** Matplotlib integrates seamlessly with other Python libraries like NumPy, Pandas, and SciPy, making it a versatile tool for data analysis and visualization.
- **Large community and extensive documentation:** Matplotlib has a large and active community, providing ample support and resources.
- **Foundation for other libraries:** Many other visualization libraries, such as Seaborn, are built on top of Matplotlib, making it a foundational tool in the Python visualization ecosystem.

SEABORN

- **Built on Matplotlib:** Seaborn is built on top of Matplotlib, inheriting its capabilities.
- **Higher-level interface:** Provides a more concise and intuitive interface for creating visually appealing plots.
- **Defaults:** Comes with built-in styles and color palettes, making it easier to create attractive visualizations without extensive customization.
- **Statistical plots:** Offers a variety of statistical plots, such as regression plots, joint plots, and pair plots.
- **Seaborn-specific plot types:** Provides plot types like catplots and relplots that are specific to Seaborn.

Advantages of Seaborn:

- **Simplified syntax:** Seaborn provides a more concise and intuitive syntax compared to Matplotlib, making it easier to learn and use.
- **Built-in styles and color palettes:** Seaborn comes with pre-defined styles and color palettes, making it easier to create visually appealing plots without extensive customization.
- **Statistical plots:** Seaborn offers a variety of statistical plots, such as regression plots, joint plots, and pair plots, that are not readily available in Matplotlib.
- **Seaborn-specific plot types:** Seaborn provides plot types like catplots and relplots that are specific to Seaborn.
- **Integration with Pandas:** Seaborn is seamlessly integrated with Pandas, allowing you to work directly with Data Frames and Series for data visualization.
- **Focus on aesthetics:** Seaborn is designed to create visually appealing plots by default, making it a good choice for presentations and publications.
- **Consistent API:** Seaborn maintains a consistent API across different plot types, making it easier to learn and use.

Key Differences

- **Customization:** Matplotlib offers more granular control over plot elements, while Seaborn provides a more streamlined approach.
- **Ease of use:** Seaborn is generally easier to learn and use for beginners due to its higher-level interface and built-in defaults.
- **Statistical plots:** Seaborn excels at creating statistical plots, while Matplotlib may require more effort for these types of visualizations.
- **Performance:** Matplotlib is typically faster for simple plots, but Seaborn can be more efficient for complex visualizations.

When to Use Which

- **Matplotlib:** Use Matplotlib when you need fine-grained control over plot elements, or for complex visualizations that require custom code.
- **Seaborn:** Use Seaborn when you want to create visually appealing plots quickly and easily, or when you need to visualize statistical relationships.

In many cases, you can effectively use both Matplotlib and Seaborn together. Seaborn can be used to create a base plot, and then Matplotlib can be used to add custom elements or fine-tune the visualization.

Note: For code please refer the jupyter notebook.