

**School of Computer Science and Engineering**

**Information Science and Technology (AI and DS)**

**DATA ANALYSIS AND VISUALIZATION**

Group Assignment on

Chapter-8

**“PLOTTING WITH SEABORN”**

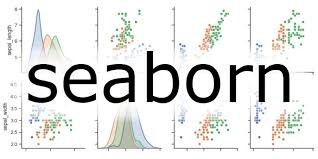
**Group Members:**

|  |  |
| --- | --- |
| **Roll numbers** | **Names** |
| 20211ISD0005 | E Rahul |
| 20211ISD0009 | Anamaneni Venkata Rohith Kumar |
| 20211ISD0020 | Pinisetty Sushmanth |
| 20211ISD0022 | Alluru Pavan Kumar Reddy |
| 20211ISD0034 | Kishan K |
| 20211ISD0035 | Medam Bala Venkata Sandeep Reddy |
| 20211ISD0037 | Muppala Prudhvi Raju |
| 20211ISD0040 | Richard Sylvester Ramesh |
| 20211ISD0041 | T Yeshwanth Teja |
| 20211ISD0043 | K Sai Karthik |

Under the guidance of:

Ms. S Poornima Asst. Prof-CSE

**Introduction to Seaborn**

Seaborn is a Python visualization library based on matplotlib that provides a high-level interface for drawing attractive statistical graphics. It's built on top of matplotlib and tightly integrated with the Pydata stack, including Pandas and NumPy. Seaborn is a statistical data visualization library. Seaborn's primary goal is to make it easy to create complex visualizations from Pandas Data Frame data, while also offering concise syntax and a plethora of customization options.

**Key Features:**

High-level Interface: Seaborn provides a simple and intuitive interface for creating complex visualizations with minimal code.

**Integration with Pandas**: Seaborn seamlessly works with Pandas Data Frames, making it easy to visualize data directly from data structures commonly used in data analysis.

**Statistical Plotting**: Seaborn includes functions for visualizing relationships in data through statistical models, making it particularly useful for exploratory data analysis.

**Attractive Aesthetics**: Seaborn comes with visually appealing default themes and colour palettes, enhancing the quality of plots without requiring manual customization.

**Flexible Customization**: While Seaborn offers sensible defaults for many plot types, it also allows for extensive customization to fine-tune plots to specific needs.

There are several tools that can be used for data visualization, but Seaborn offers several advantages. First, Seaborn's main purpose is to make data visualization easy. It was built to automatically handle a lot of complexity behind the scenes. Second, Seaborn works extremely well with pandas data structures. Pandas is a Python library that is widely used for data analysis

**Installation:**

You can install seaborn using pip :

pip install seaborn

**Importing Seaborn:**

Once installed, you can import Seaborn as follows:

import seaborn as sns

**Loading Datasets:**

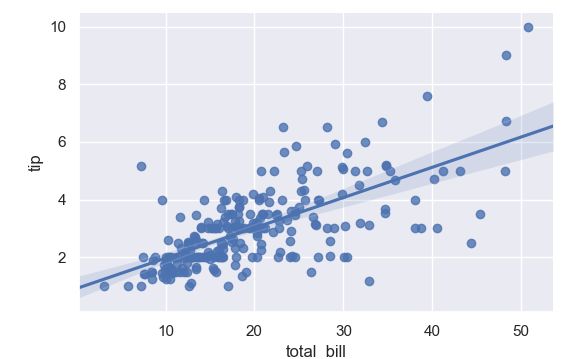
Seaborn comes with built-in datasets to demonstrate its functionality. For example, let's load the Iris dataset:

iris = sns.load\_dataset('iris')

**Basic Plotting:**

Seaborn makes it easy to create attractive plots with just a few lines of code. Here's an example of a scatter plot:

sns.scatterplot(x='sepal\_length', y='sepal\_width', data=iris)



**Statistical Relationships:**

Seaborn allows you to visualize statistical relationships between variables. For instance, you can create a linear regression plot:

sns.lmplot(x='sepal\_length', y='sepal\_width', data=iris)

**Distribution Plots:**

Seaborn provides various functions to visualize the distribution of data. Here's a histogram with a kernel density estimate (KDE):

sns.histplot(data=iris, x='sepal\_length', kde=True)

Distribution plots in Seaborn are used to visualize the distribution of univariate data and to understand its underlying structure. These plots help in analyzing the central tendency, variability, and skewness of a dataset.



**Parameters:**

**Variables:**

**x, y:** Names of variables in the DataFrame to be plotted on the x and y axes, respectively.

**Hue:** Variable in the DataFrame to map plot aspects to different colors. It adds color differentiation based on categories.

**Plot Appearance:**

**Palette:** Color palette to use for coloring the plot elements. Seaborn offers various built-in palettes and the ability to create custom palettes.

**color:** Color specification for plot elements, overriding the default palette.

**linewidth, line style, alpha**: Control the width, style, and transparency of lines in plots.

**marker, marker size:** Specify the marker style and size for scatter plots

**Setting Up Seaborn:**

Setting up Seaborn involves configuring various aspects of its appearance and behaviour to suit your preferences and requirements. Seaborn provides several options for customization, ranging from setting default themes to modifying plot aesthetics

**Setting Default Themes:**

Seaborn comes with several built-in themes that control the appearance of plots, including colours, fonts, gridlines, and more. You can set a default theme using sns.set\_theme():

import seaborn as sns

# Set default theme

sns.set\_theme()

Seaborn offers different themes such as "darkgrid", "whitegrid", "dark", "white", and "ticks". You can specify a theme by passing it as an argument to sns.set\_theme().

**Customizing Aesthetics:**

Seaborn allows you to customize various plot aesthetics, such as colors, styles, and font sizes, to match your preferences. You can use Seaborn's built-in color palettes or create custom palettes:

# Customizing color palette

custom\_palette = ["#FF5733", "#33FFC7", "#337BFF"]

sns.set\_palette(custom\_palette)

Additionally, you can adjust other aesthetics like line width (sns.set\_context()) and font scale (sns.set\_context()), among others.

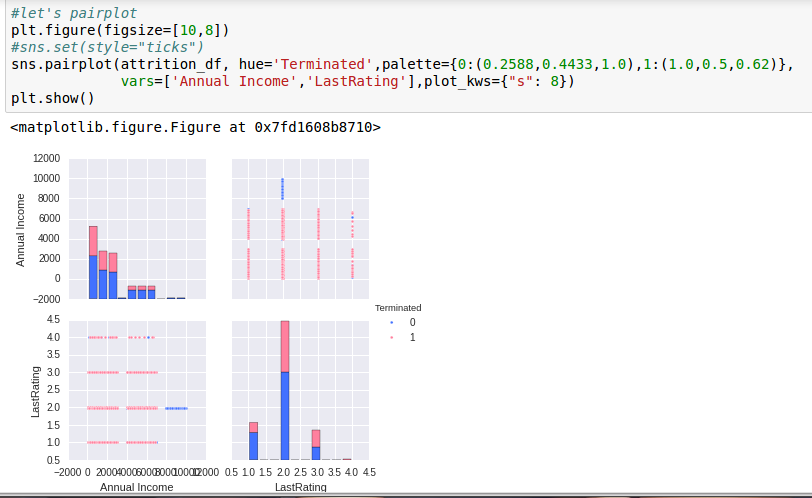
**Adjusting Plot Size:**

You can control the size of Seaborn plots using Matplotlib's plt.figure() function. This allows you to specify the width and height of the plot in inches:

import matplotlib.pyplot as plt

# Adjust plot size

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

**Controlling Plot Components:**

Seaborn provides functions to customize specific plot components, such as gridlines, axes labels, titles, and legends. For example, you can adjust the visibility and appearance of gridlines

# Hide gridlines

sns.set\_style("white") # or "ticks" to show minor ticks

**Enabling or Disabling Context:**

Seaborn's context settings control various scaling parameters that affect the size of elements in the plot. You can adjust the context using sns.set\_context(), which allows you to specify the context ("paper", "notebook", "talk", or "poster") and optionally the font scale:

# Set context to 'talk' with larger font scale

sns.set\_context("talk", font\_scale=1.2)

**Using Dark Mode:**

Seaborn also supports dark mode themes, which can be enabled using the dark or darkgrid theme:

# Enable dark mode

sns.set\_theme(style="dark")

**Resetting to Defaults:**

If you want to reset all settings to their default values, you can use sns.reset\_defaults();

# Reset to default settings

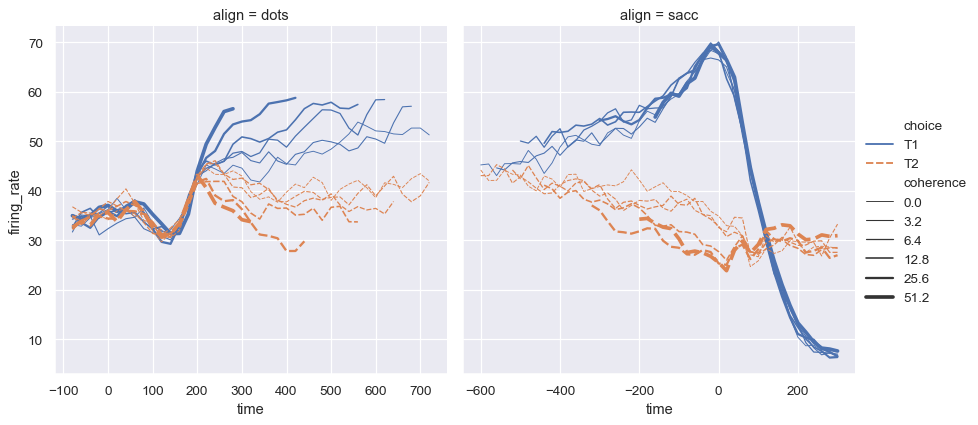
sns.reset\_defaults()

**Working with Pandas Data Frames:**

Seaborn is designed to work seamlessly with Pandas Data Frames, making it easy to visualize data directly from data structures commonly used in data analysis. Before creating plots with Seaborn, ensure that you have imported the necessary libraries, including Pandas:

**Configuring Matplotlib Backend:**

Seaborn relies on Matplotlib for plotting, so it's essential to configure the Matplotlib backend appropriately. If you're working in a Jupyter Notebook, you can use %matplotlib inline or %matplotlib notebook to display plots inline or interactively, respectively.

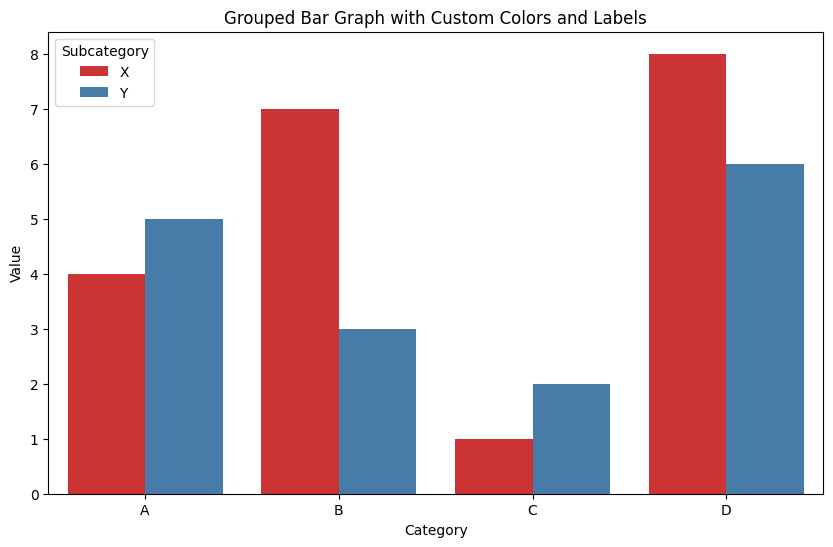
**Configuring Matplotlib Backend:** Seaborn relies on Matplotlib for plotting, so it's essential to configure the Matplotlib backend appropriately. %matplotlib notebook to display plots inline or interactively, respectively.

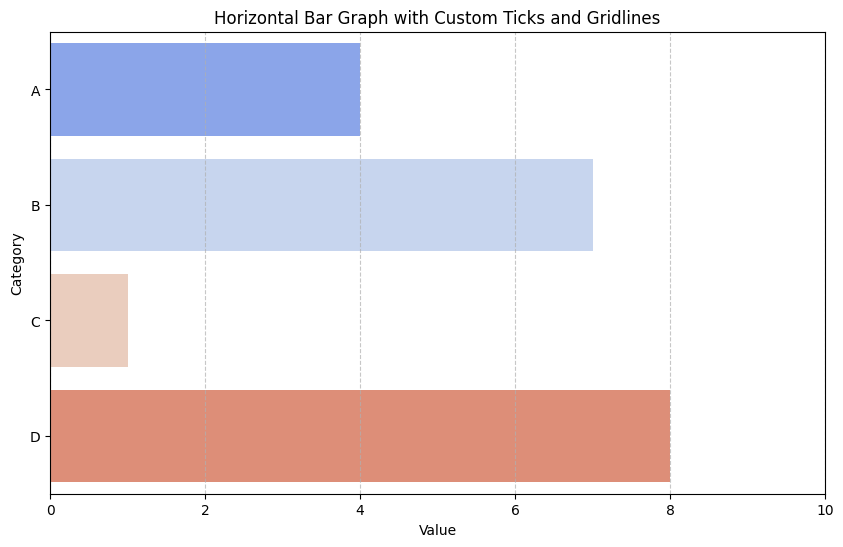
**Basic Plots**

**SEABRON BAR PLOTS**

**Grouped Bar Graph with Custom Colors and Labels**

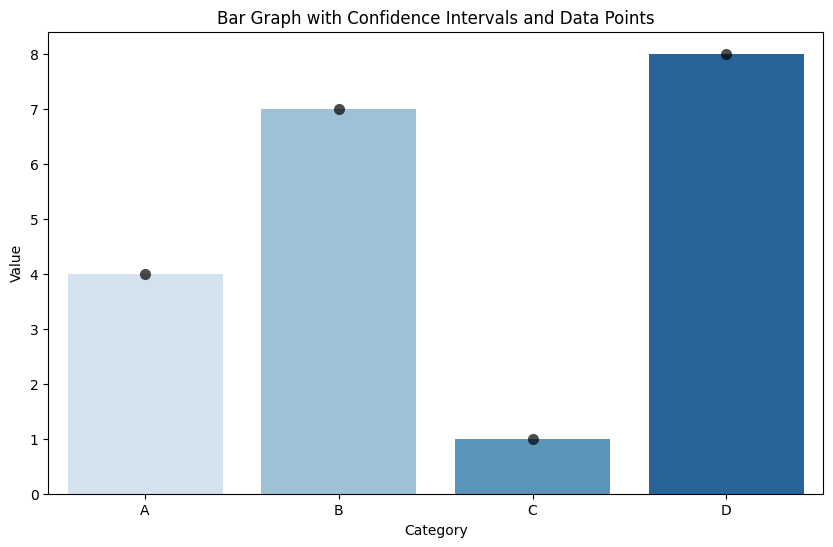
This graph demonstrates a grouped bar plot with custom colors using the Set1 palette. It includes subcategories within each main category, offering a clear comparison between different subcategories. The legend, positioned to the right, helps differentiate between subcategories, and the axes are labeled for clarity.

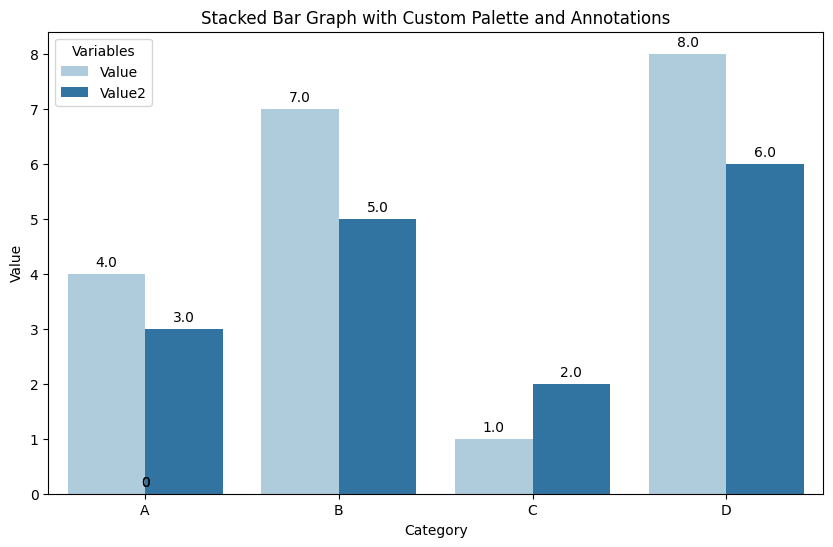
**Horizontal Bar Graph with Custom Ticks and Gridlines**

This horizontal bar graph uses a coolwarm palette and custom x-ticks set at intervals of 2. The x-axis gridlines are styled with a dashed line and slight transparency for a clean, professional look. This graph style is particularly useful for emphasizing differences between categories when the category labels are longer.

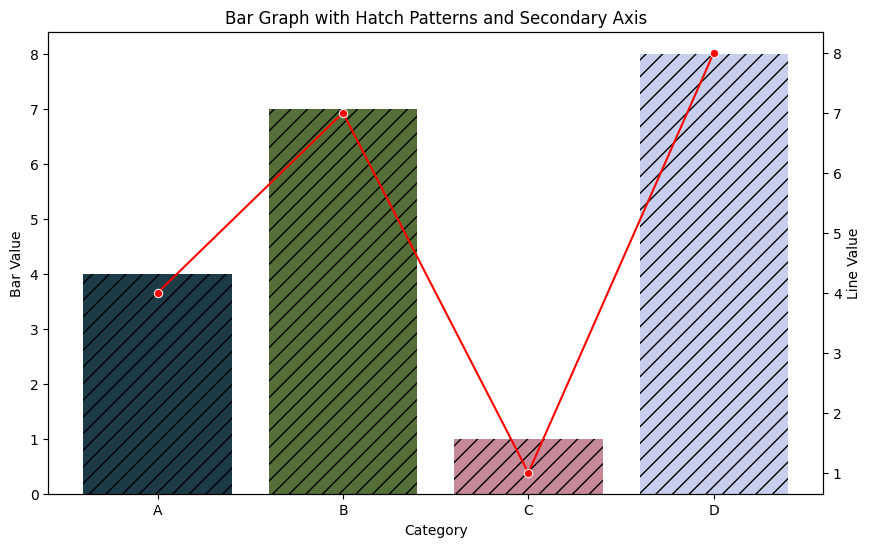
**Bar Graph with Confidence Intervals and Data Points**

This bar graph combines bar plots with confidence intervals (95%) and individual data points using stripplot. The Blues palette gives a gradient color effect, while the black data points with slight jitter provide insight into the distribution within each category. This combination is effective for visualizing central tendencies along with variability in data.

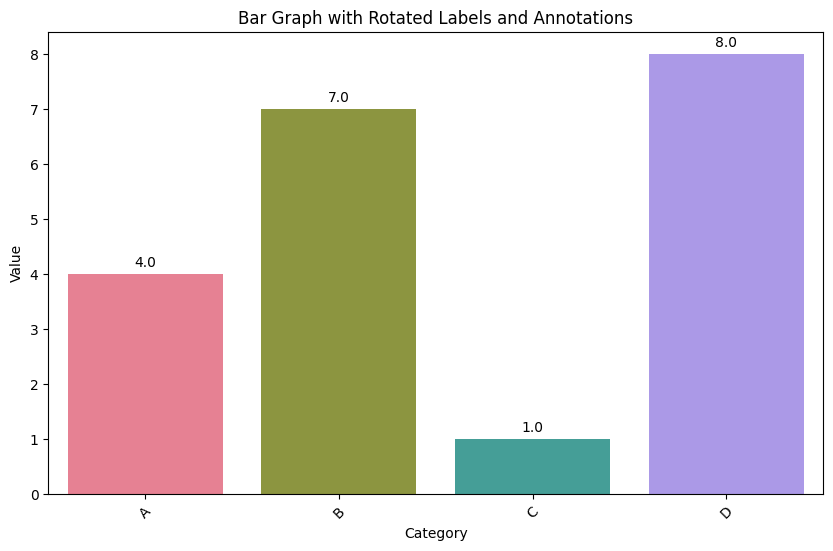
**Stacked Bar Graph with Custom Palette and Annotations**

This stacked bar graph uses the Paired palette to distinguish between two variables, ‘Value’ and ‘Value2’. Annotations on each bar show the precise values, enhancing readability. The legend indicates the variables, and the x and y-axis labels provide context. Stacked bar grape useful for comparing the composition of categories.

**Bar Graph with Hatch Patterns and Secondary Axis**

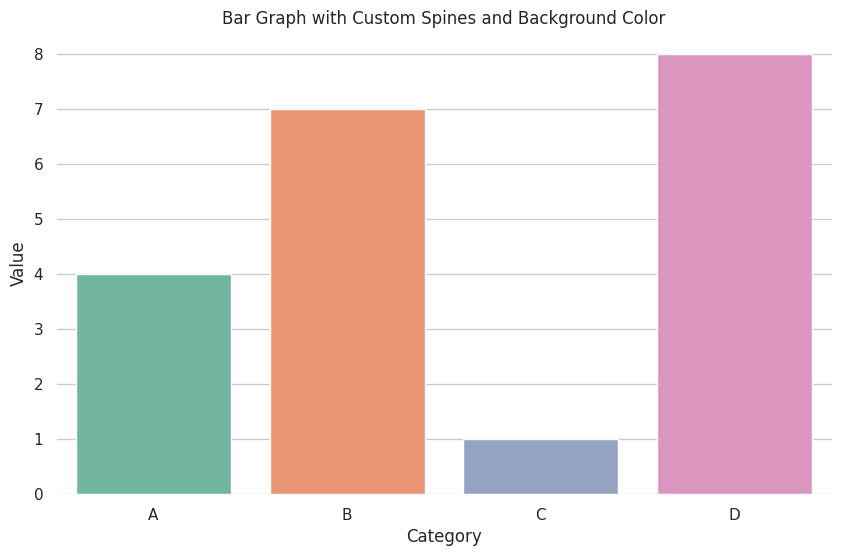
This bar graph incorporates hatch patterns and a secondary y-axis. The primary y-axis shows bar values with a cubehelix palette and // hatching. The secondary y-axis (on the right) displays a line plot with markers, providing a different perspective on the data. This combination is beneficial when you need to compare two related datasets with different scales.

**Bar Graph with Rotated Labels and Annotations**

This bar graph features rotated x-axis labels (45 degrees) for better readability, especially useful when dealing with long category names. The husl palette provides distinct colors for each bar. Annotations on top of each bar display the exact values, making it easier to interpret the data at a glance.

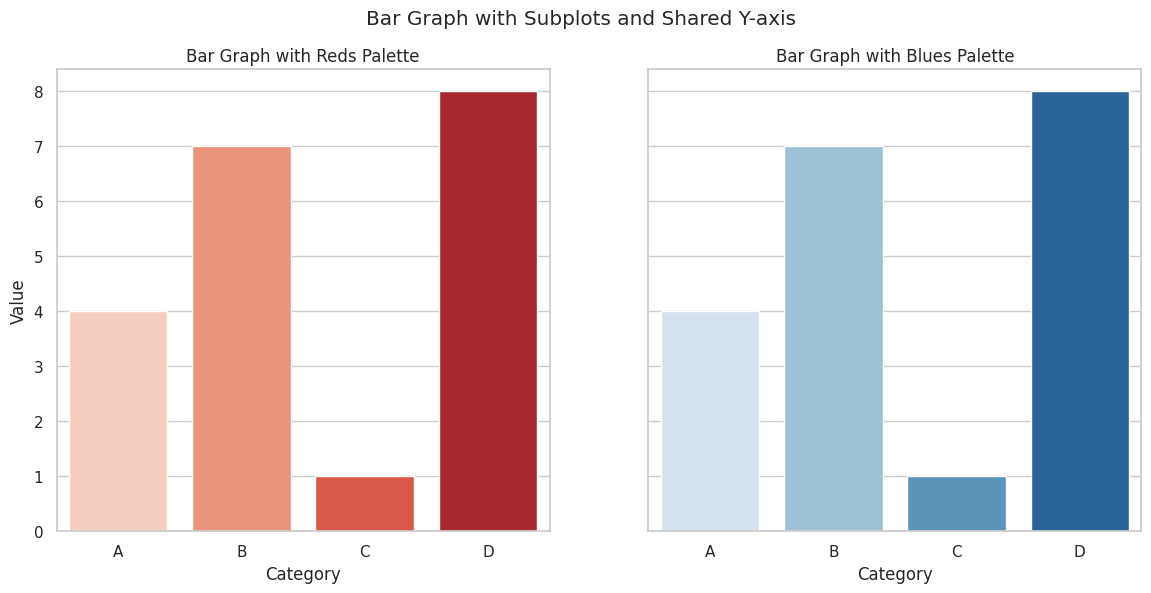
**Bar Graph with Custom Spines and Background Color**

This bar graph uses a white grid background and custom spine colors. The Set2 palette provides visually appealing colors. The spines (axes lines) are customized for a cleaner look, with the left and bottom spines colored in gray. Despining (removing top and right spines) further enhances the minimalist aesthetic.



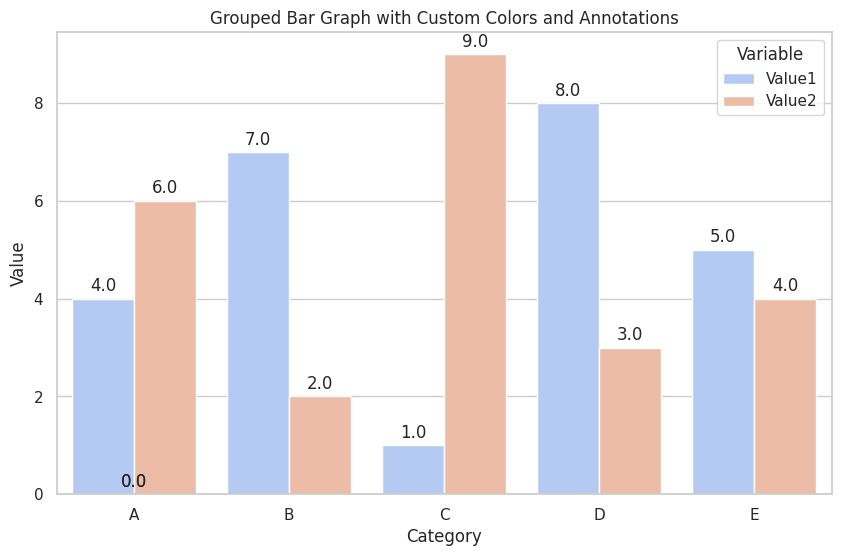
**Bar Graph with Subplots and Shared Y-axis**

This bar graph utilizes the ch:2.5,.25 color palette for density shading, providing a gradient effect within the bars. Custom legend labels (‘Low’, ‘Medium’, ‘High’, ‘Very High’) are manually set, aiding in interpreting the density levels. This style is effective for representing hierarchical data within categories.

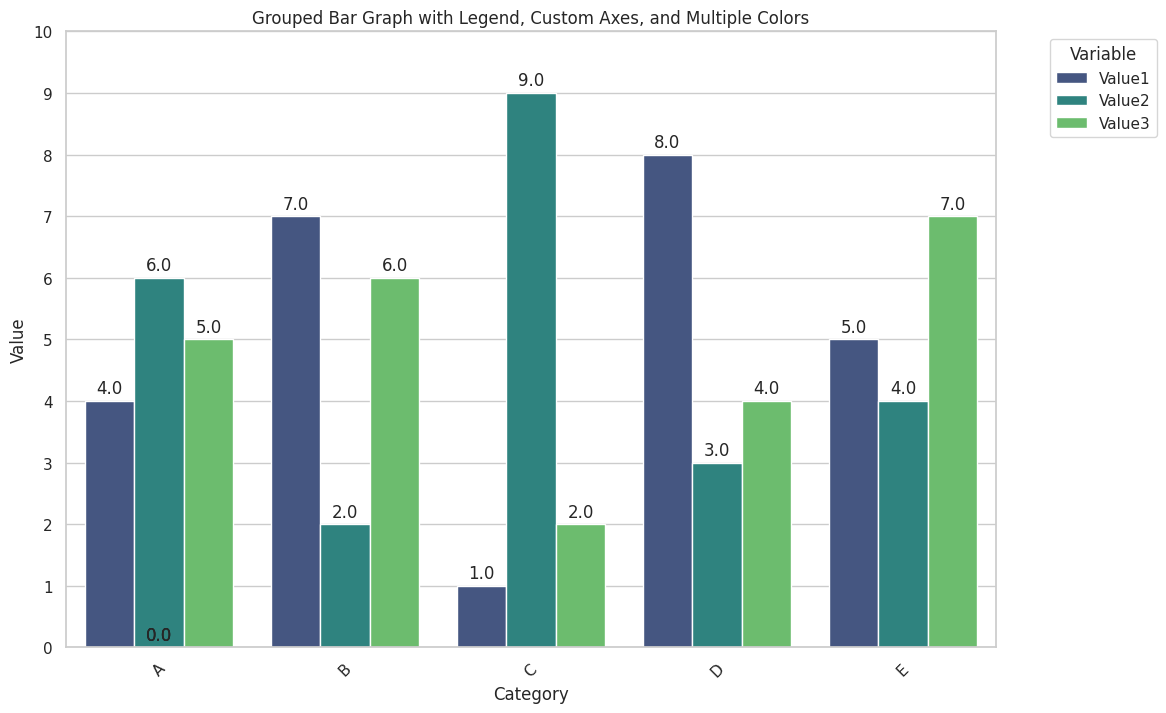


**Grouped Bar Graph with Custom Colors and Annotations**

This grouped bar graph visualizes the values of two variables (Value1 and Value2) across five categories (A, B, C, D, E). The dataset is first converted to a long format using pd.melt to facilitate grouped plotting with the hue parameter. The coolwarm palette provides a visually appealing contrast between the two variables. Each bar is annotated with its corresponding value for precise data interpretation. The plot includes a title and axis labels for context. Grouped bar graphs are useful for comparing multiple variables across different categories, providing a clear, side-by-side comparison.



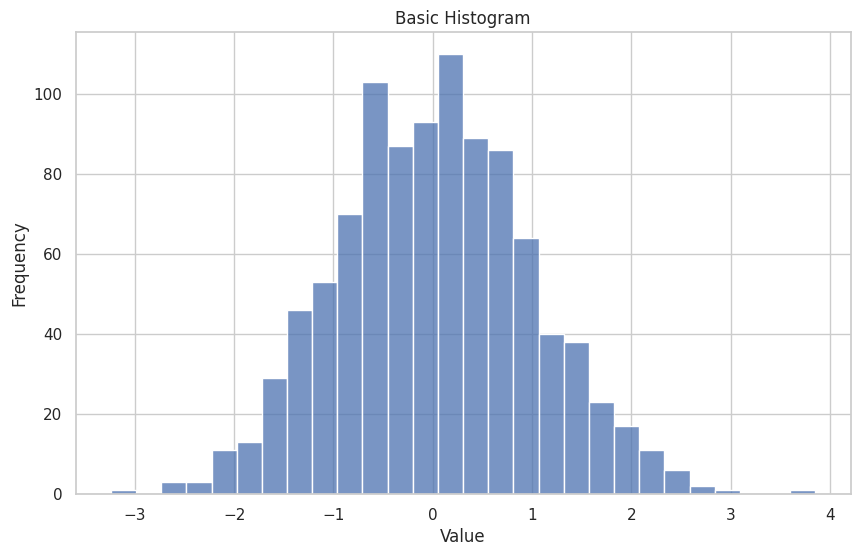
**Grouped Bar Graph with Legend, Custom Axes, and Multiple Colours**

This grouped bar graph visualizes three variables (Value1, Value2, Value3) across five categories (A, B, C, D, E). The dataset is melted to a long format to facilitate grouped plotting with the hue parameter. The viridis palette offers a diverse and vibrant color scheme for the bars. Each bar is annotated with its corresponding value for clear data interpretation. The x-axis labels are rotated for better readability, and the y-axis is customized to show ticks at every integer value up to 10. The legend is positioned outside the plot area for clarity, with the title ‘Variable’. This plot is ideal for comparing multiple variables across different categories, providing a clear and detailed visual comparison.

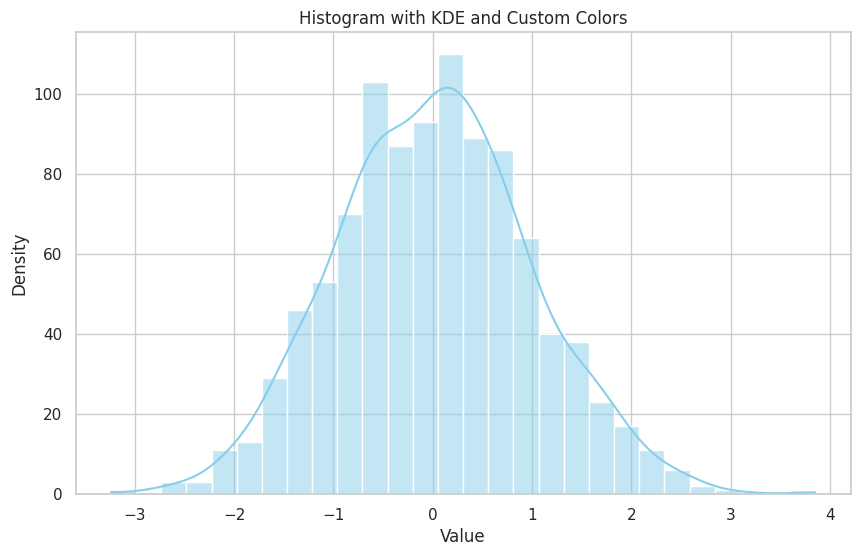
**SEABORN HISTOGRAMS**

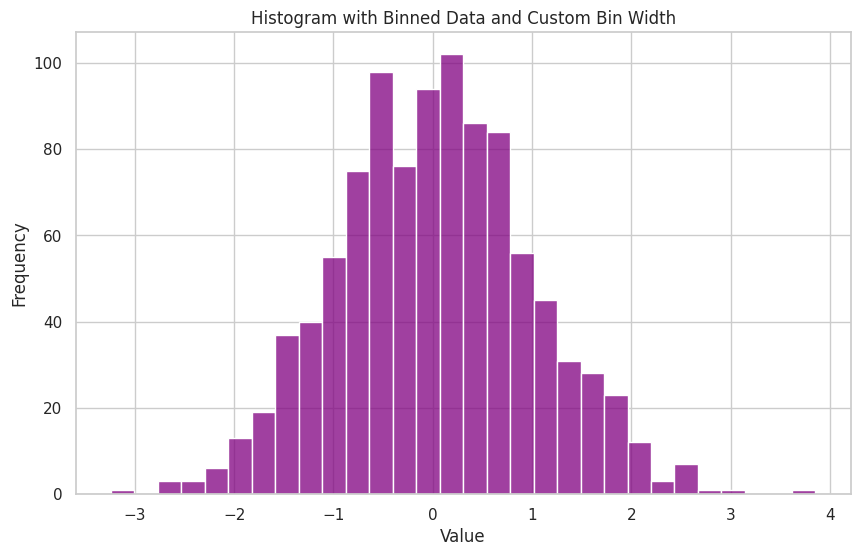
**Basic Histogram with Default Settings**

This is a basic histogram using Seaborn’s histplot function with default settings. The histogram displays the frequency distribution of the data. The x-axis represents the values, while the y-axis represents the frequency. This plot is suitable for a quick overview of the data distribution.

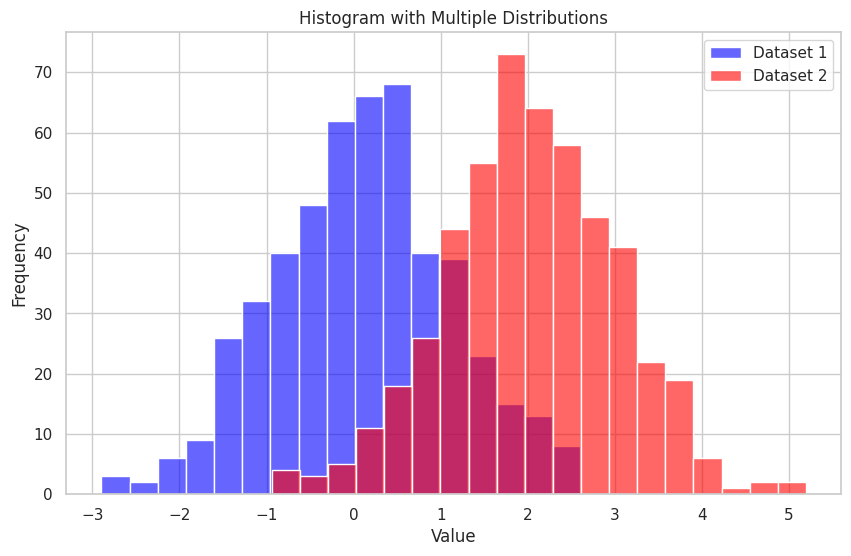
**Histogram with KDE and Custom Colours**

This histogram includes a Kernel Density Estimate (KDE) overlay, which provides a smooth curve representing the data distribution. The colour is customized to ‘sky-blue’ for both the histogram and the KDE curve. The combination of histogram bars and KDE curve offers a comprehensive view of the data density.

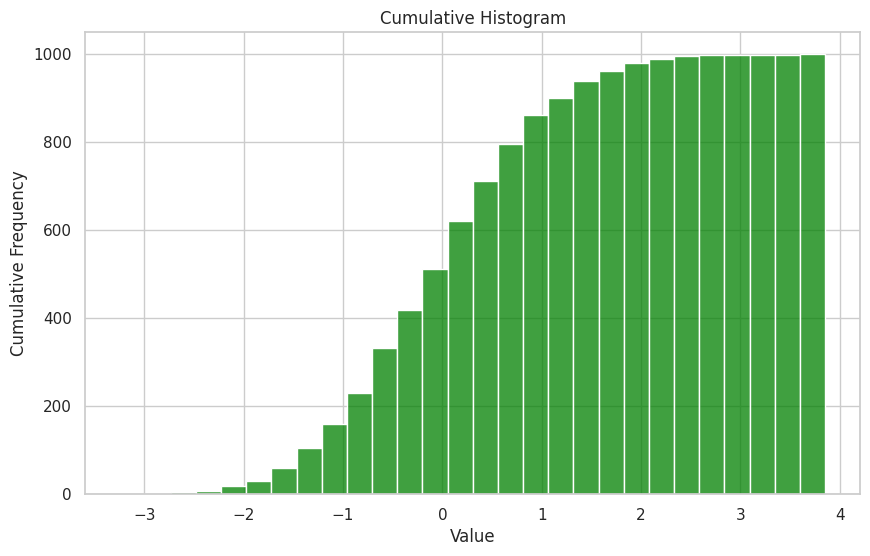
**Histogram with Binned Data and Custom Bin Width**

This histogram specifies the number of bins (30) to adjust the bin width, providing a more granular view of the data distribution. The color ‘purple’ enhances visual appeal. Custom bin widths help in better understanding the distribution by controlling the level of detail.

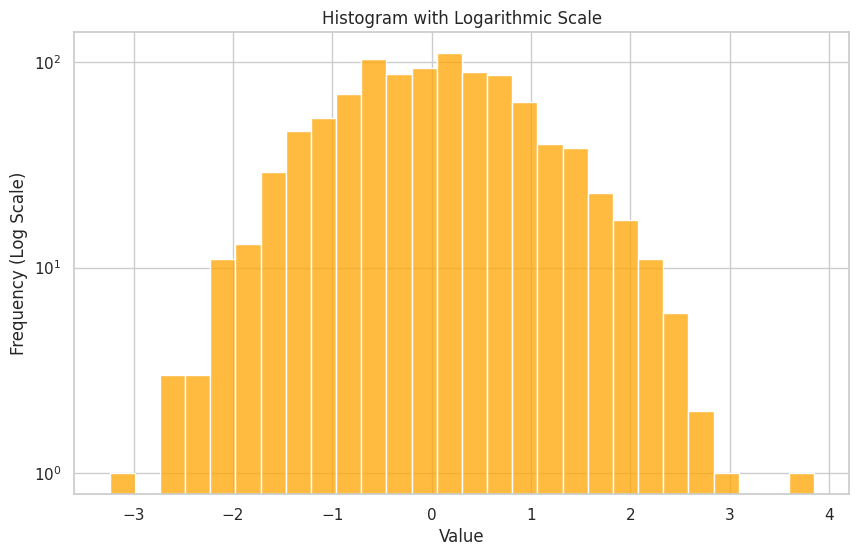
**Histogram with Multiple Distributions**

This histogram overlays two different distributions on the same plot. Dataset 1 is shown in blue, and Dataset 2 in red, with some transparency (alpha=0.6) to allow overlap visualization. The legend distinguishes between the two datasets. This type of plot is useful for comparing distributions directly.

**Cumulative Histogram**

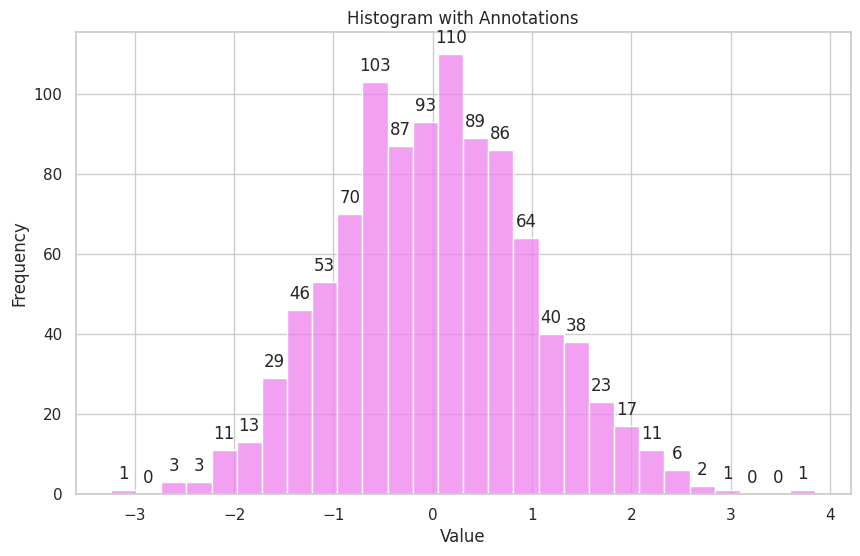
This cumulative histogram shows the cumulative frequency of the data, indicating the running total of frequencies up to each bin. The green color highlights the cumulative trend. Cumulative histograms are beneficial for understanding the proportion of data below certain values.

**Histogram with Logarithmic Scale**

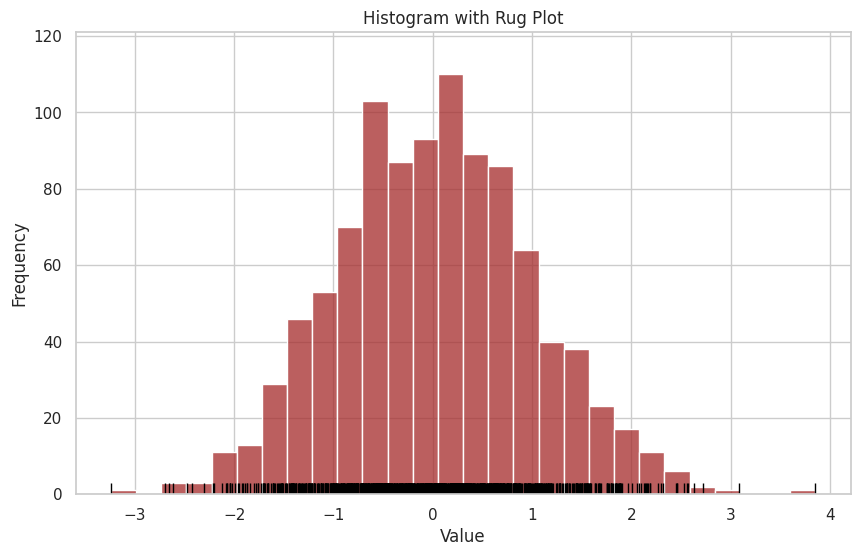
This histogram uses a logarithmic scale for the y-axis, which is effective for visualizing data with a wide range of frequencies. The orange color makes the plot visually appealing. Logarithmic scales help in emphasizing differences in the lower-frequency bins.

**Histogram with Annotations**

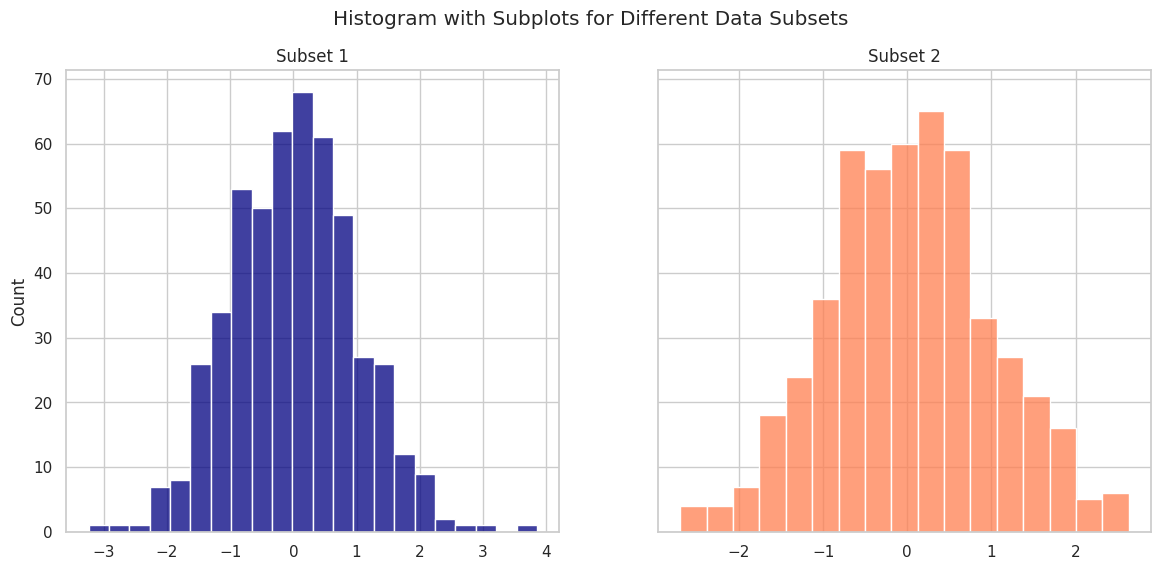
This histogram includes annotations on each bar, showing the exact frequency values. The color ‘violet’ adds a vibrant look. Annotating histograms is useful for precise data communication, especially when presenting to an audience or in reports.



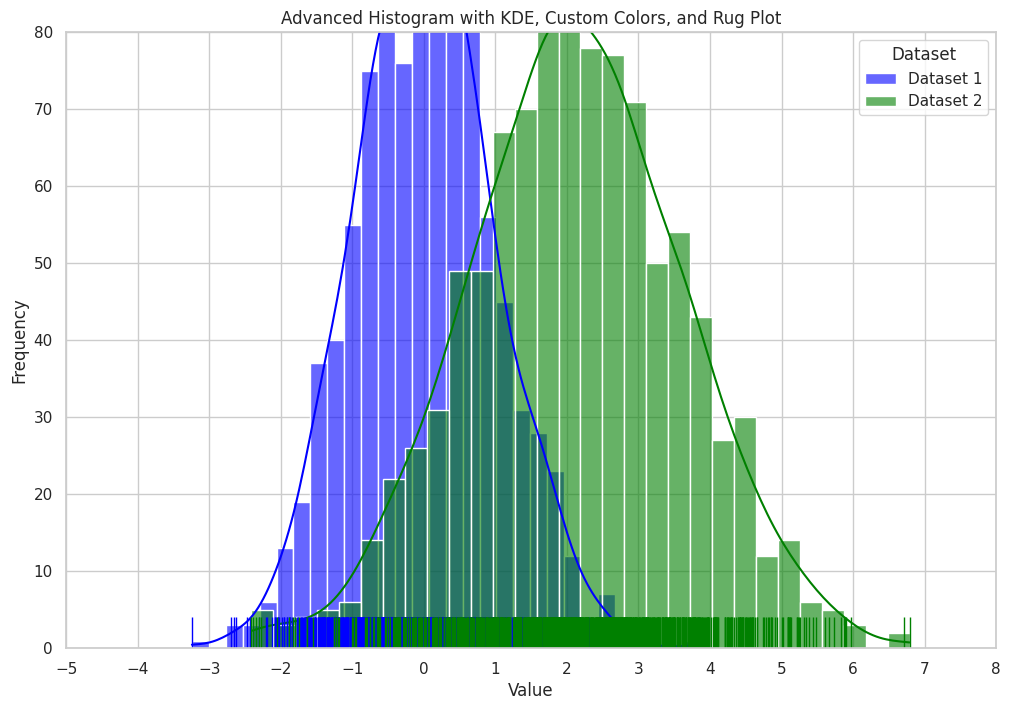
**Histogram with Rug Plot**

This histogram includes a rug plot at the bottom, showing individual data points as small ticks. The combination of a brown histogram and black rug plot provides a detailed view of the data distribution and individual observations. Rug plots are helpful for visualizing the exact values contributing to the distribution.

**Histogram with Subplots for Different Data Subsets**

This plot displays histograms for two different data subsets side by side using subplots. Subset 1 is shown in navy, and Subset 2 in coral. The shared y-axis ensures consistent frequency comparison. Subplots are effective for comparing different segments of a dataset while maintaining a clear visual distinction.

**Histogram with KDE, Custom Colours, and Rug Plot**

This advanced histogram plot visualizes the distribution of two datasets (data1 and data2). The histograms for both datasets are overlaid with Kernel Density Estimate (KDE) curves to show the smoothed distribution. The histplot function is used to plot the histograms, with custom colours (blue and green) and transparency (alpha=0.6) for better visual distinction This plot is ideal for comparing the distributions of two datasets, providing both frequency and density information along with individual data point visualization.

**SEABORN LINE PLOTS**

**Line plots are useful for visualizing data trends over intervals.**

import seaborn as sns

import matplotlib.pyplot as plt

# Example dataset

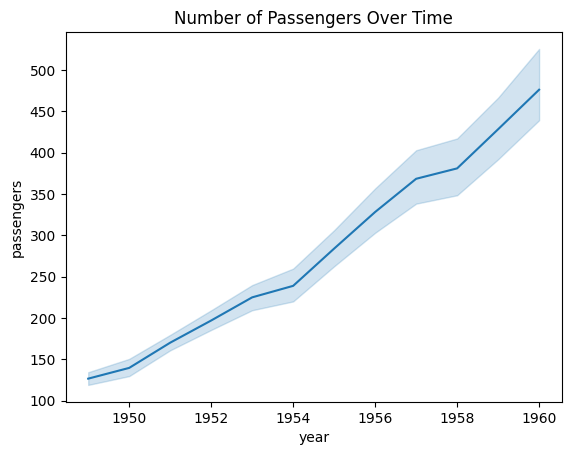
data = sns.load\_dataset("flights")

# Line plot

sns.lineplot(data=data, x="year", y="passengers")

plt.title("Number of Passengers Over Time")

plt.show()



**1. Simple Line Plot**

A simple line plot displays a single line representing data points connected over an interval, often time. It's useful for showing the overall trend of the data.

import seaborn as sns

import matplotlib.pyplot as plt

# Example dataset

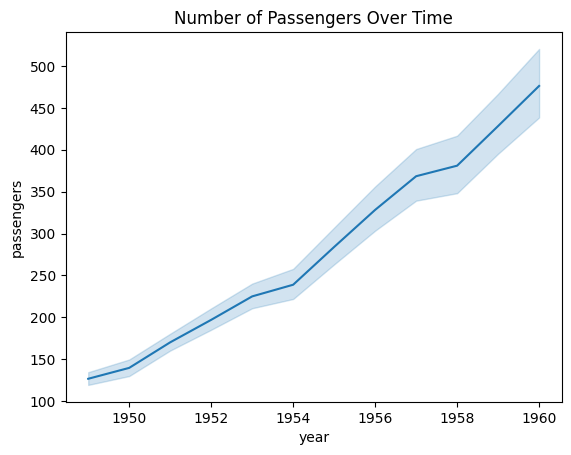
data = sns.load\_dataset("flights")

# Simple line plot

sns.lineplot(data=data, x="year", y="passengers")

plt.title("Number of Passengers Over Time")

plt.show()



**2. Multiple Lines**

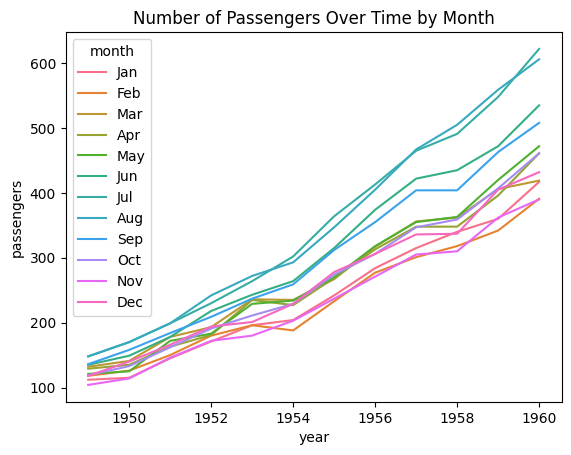
Multiple lines can represent different categories within the data, allowing comparison of trends across these categories.

# Multiple lines by month

sns.lineplot(data=data, x="year", y="passengers", hue="month")

plt.title("Number of Passengers Over Time by Month")

plt.show()



**3. Customizing Line Styles**

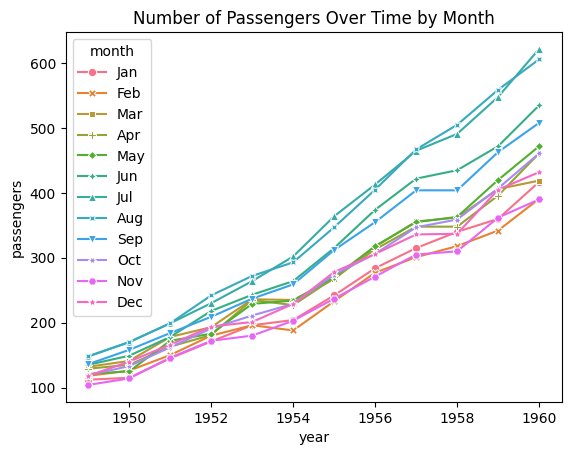
Line styles, including markers and dashes, can be customized to enhance the clarity and aesthetics of the plot.

# Line plot with markers and different line styles

sns.lineplot(data=data, x="year", y="passengers", hue="month", style="month", markers=True, dashes=False)

plt.title("Number of Passengers Over Time by Month")

plt.show()



**Scatter Plots**

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

import seaborn as sns

import matplotlib.pyplot as plt

# Example dataset

data = sns.load\_dataset("iris")

# Simple scatter plot

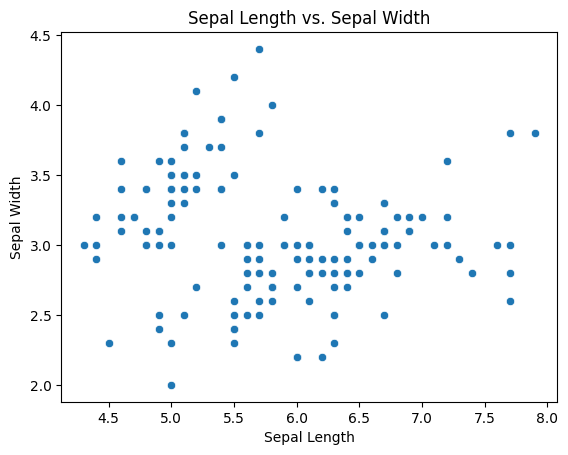
sns.scatterplot(data=data, x="sepal\_length", y="sepal\_width")

plt.title("Sepal Length vs. Sepal Width")

plt.xlabel("Sepal Length")

plt.ylabel("Sepal Width")

plt.show()



**1. Simple Scatter Plot**

A simple scatter plot displays individual data points, providing a clear view of their distribution and potential correlations between two variables.

# Example dataset

data = sns.load\_dataset("iris")

# Simple scatter plot

sns.scatterplot(data=data, x="sepal\_length", y="sepal\_width")

plt.title("Sepal Length vs. Sepal Width")

plt.show()

**2. Scatter Plot with Hues**

Using hues allows differentiation of data points based on a third categorical variable, enhancing the plot’s informational content

# Scatter plot with different colors for each species

sns.scatterplot(data=data, x="sepal\_length", y="sepal\_width", hue="species")

plt.title("Sepal Length vs. Sepal Width by Species")

plt.show()



**3. Customizing Markers and Sizes**

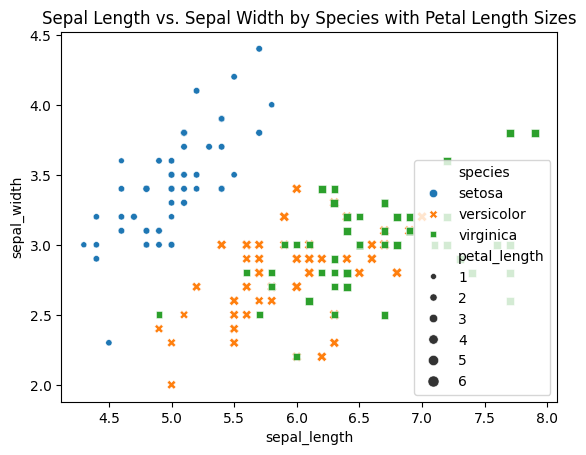
Different markers can be used for different categories, making it easier to distinguish between groups visually.

# Scatter plot with different markers and sizes

sns.scatterplot(data=data, x="sepal\_length", y="sepal\_width", hue="species", style="species", size="petal\_length")

plt.title("Sepal Length vs. Sepal Width by Species with Petal Length Sizes")

plt.show()



**SEABORN BOX PLOTS**

Box plots (or box-and-whisker plots) are used to visualize the distribution of a dataset by displaying the dataset’s quartiles and averages.

**1. Simple Box Plot**

Description: A simple box plot shows the median, quartiles, and potential outliers for a single variable, providing a clear summary of its distribution

import seaborn as sns

import matplotlib.pyplot as plt

# Example dataset

data = sns.load\_dataset("iris")

# Simple box plot

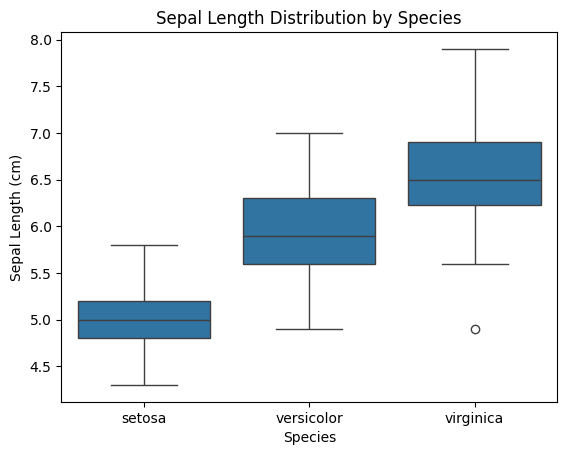
sns.boxplot(x=data["species"], y=data["sepal\_length"])

plt.title("Sepal Length Distribution by Species")

plt.xlabel("Species")

plt.ylabel("Sepal Length (cm)")

plt.show()



**2. Box Plot with Hues**

Description: Box plots with hues can compare distributions across multiple categories, such as different species or groups

# Box plot with hues

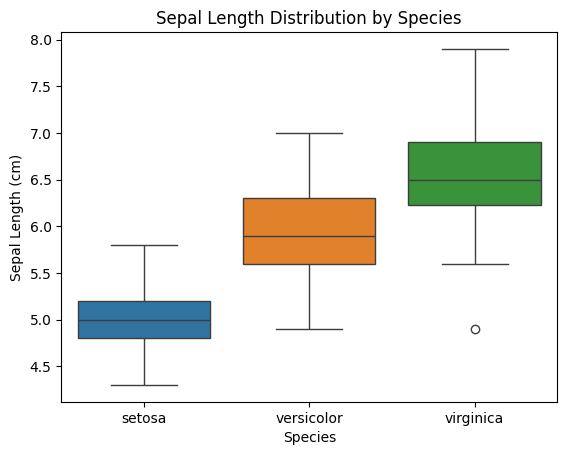
sns.boxplot(data=data, x="species", y="sepal\_length", hue="species")

plt.title("Sepal Length Distribution by Species")

plt.xlabel("Species")

plt.ylabel("Sepal Length (cm)")

plt.show()



**3. Box Plot with Multiple Categories**

Description: Box plots can be used to display distributions for multiple categories, such as comparing distributions across different groups or conditions.

# Example dataset

data = sns.load\_dataset("tips")

# Box plot with multiple categories

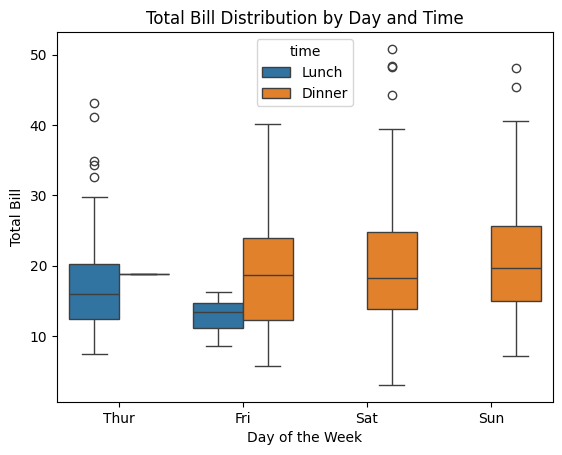
sns.boxplot(data=data, x="day", y="total\_bill", hue="time")

plt.title("Total Bill Distribution by Day and Time")

plt.xlabel("Day of the Week")

plt.ylabel("Total Bill")

plt.show()



**Customizing Colors and Palettes**

You can use a variety of predefined color palettes or define your own custom palette.

**1. Using a Predefined Palette**

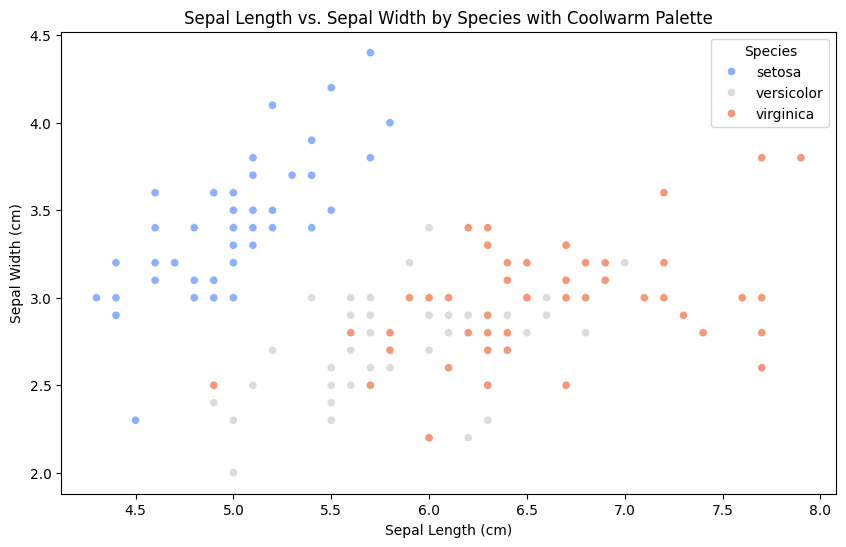
Scatter Plot with a Predefined Palette Description: This plot visualizes the relationship between sepal length and sepal width of different iris species using a predefined color palette.

Customizations:

Palette: "coolwarm" is used to differentiate between species.

Title: "Sepal Length vs. Sepal Width by Species with Coolwarm Palette"

Labels: X-axis labeled "Sepal Length (cm)" and Y-axis labeled "Sepal Width (cm)"

Legend: Legend titled "Species" to identify different species.

**2. Using a Custom Palette**

# Define a custom palette

custom\_palette = {

"setosa": "#FF6347", # Tomato

"versicolor": "#4682B4", # Steel Blue

"virginica": "#32CD32" # Lime Green

}

# Scatter plot with a custom palette

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

sns.scatterplot(data=data, x="sepal\_length", y="sepal\_width", hue="species", palette=custom\_palette)

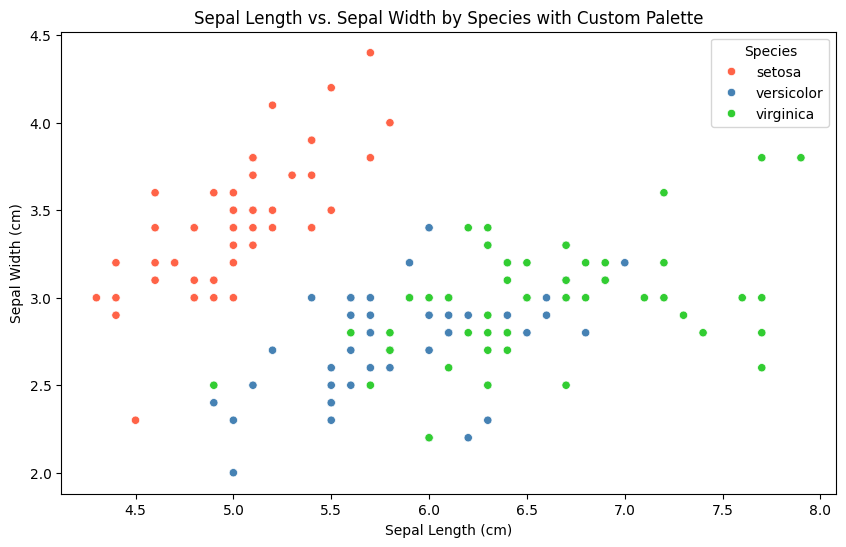
plt.title("Sepal Length vs. Sepal Width by Species with Custom Palette")

plt.xlabel("Sepal Length (cm)")

plt.ylabel("Sepal Width (cm)")

plt.legend(title="Species")

plt.show()



**Customizing Markers**

You can customize the markers used in the scatter plot to distinguish between categories.

Scatter Plot with Custom Markers Description: This plot uses different markers to represent different iris species, enhancing visual differentiation.

Customizations:

Markers: Different markers ("o", "s", "D") are used for each species.

Title: "Sepal Length vs. Sepal Width by Species with Custom Markers"

Labels: X-axis labeled "Sepal Length (cm)" and Y-axis labeled "Sepal Width (cm)"

Legend: Legend titled "Species" to identify different species.

# Scatter plot with custom markers

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

sns.scatterplot(data=data, x="sepal\_length", y="sepal\_width", hue="species", style="species", markers=["o", "s", "D"])

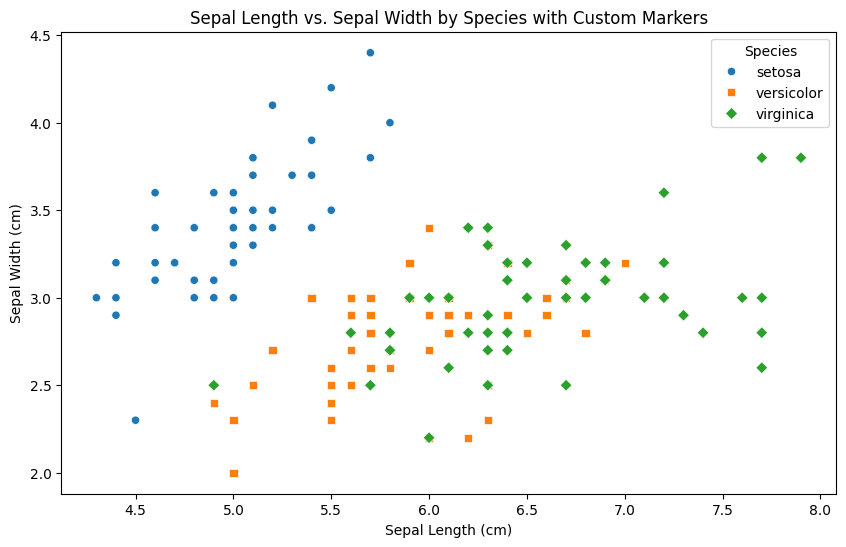
plt.title("Sepal Length vs. Sepal Width by Species with Custom Markers")

plt.xlabel("Sepal Length (cm)")

plt.ylabel("Sepal Width (cm)")

plt.legend(title="Species")

plt.show()



**Customizing Point Sizes**

You can adjust the sizes of the points based on another variable.

Scatter Plot with Size Variation Description: This plot varies the size of the points based on the petal length, providing an additional dimension to the visualization.

Customizations:

Size: Points are sized based on petal length, with sizes ranging from 20 to 200.

Palette: "viridis" palette is used to color the points.

Title: "Sepal Length vs. Sepal Width by Species with Petal Length Sizes"

Labels: X-axis labeled "Sepal Length (cm)" and Y-axis labeled "Sepal Width (cm)"

Legend: Legend titled "Species" to identify different species.

# Scatter plot with size variation

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

sns.scatterplot(data=data, x="sepal\_length", y="sepal\_width", hue="species", size="petal\_length", sizes=(20, 200), palette="viridis")

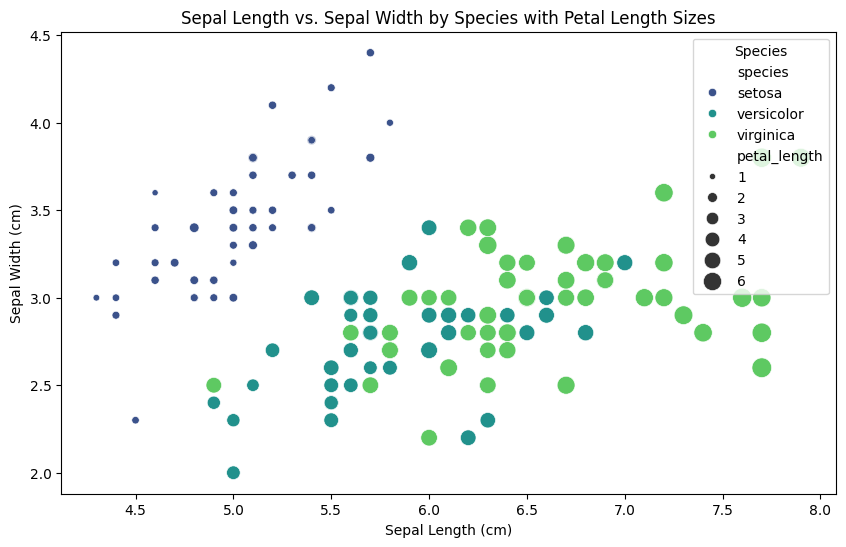
plt.title("Sepal Length vs. Sepal Width by Species with Petal Length Sizes")

plt.xlabel("Sepal Length (cm)")

plt.ylabel("Sepal Width (cm)")

plt.legend(title="Species")

plt.show()



**Adding Regression Line**

For linear relationships, you can add a regression line using sns.lmplot().

Scatter Plot with Regression Line Description: This plot includes a regression line to show the linear relationship between sepal length and sepal width for each species.

Customizations:

Regression Line: sns.lmplot() is used to add a regression line.

Markers: Different markers ("o", "s", "D") are used for each species.

Palette: "muted" palette is used to color the points.

Title: "Sepal Length vs. Sepal Width with Regression Line"

Labels: X-axis labeled "Sepal Length (cm)" and Y-axis labeled "Sepal Width (cm)"

Legend: Legend titled "Species" to identify different species.

# Scatter plot with regression line

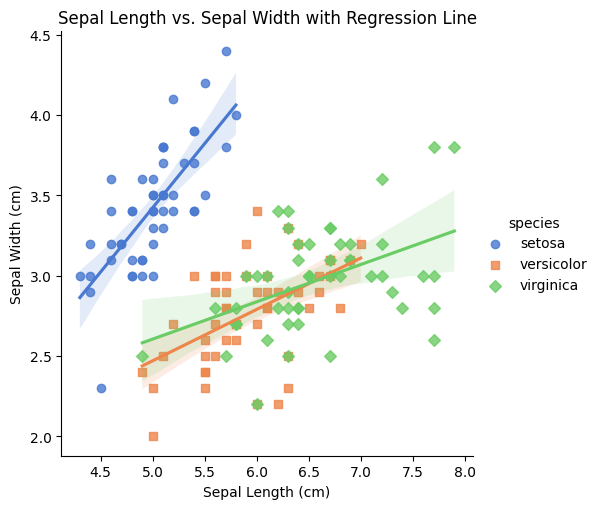
sns.lmplot(data=data, x="sepal\_length", y="sepal\_width", hue="species", markers=["o", "s", "D"], palette="muted")

plt.title("Sepal Length vs. Sepal Width with Regression Line")

plt.xlabel("Sepal Length (cm)")

plt.ylabel("Sepal Width (cm)")

plt.show()



**Combining Multiple Customizations**

You can combine different customization options to create a highly informative and visually appealing scatter plot.

Scatter Plot with Regression Line Description: This plot includes a regression line to show the linear relationship between sepal length and sepal width for each species.

Customizations:

Regression Line: sns.lmplot() is used to add a regression line.

Markers: Different markers ("o", "s", "D") are used for each species.

Palette: "muted" palette is used to color the points.

Title: "Sepal Length vs. Sepal Width with Regression Line"

Labels: X-axis labeled "Sepal Length (cm)" and Y-axis labeled "Sepal Width (cm)"

Legend: Legend titled "Species" to identify different species.

# Combined customizations

plt.figure(figsize=(12, 8))

sns.scatterplot(

data=data,

x="sepal\_length",

y="sepal\_width",

hue="species",

style="species",

size="petal\_length",

sizes=(50, 250),

palette="dark",

markers=["o", "s", "D"]

)

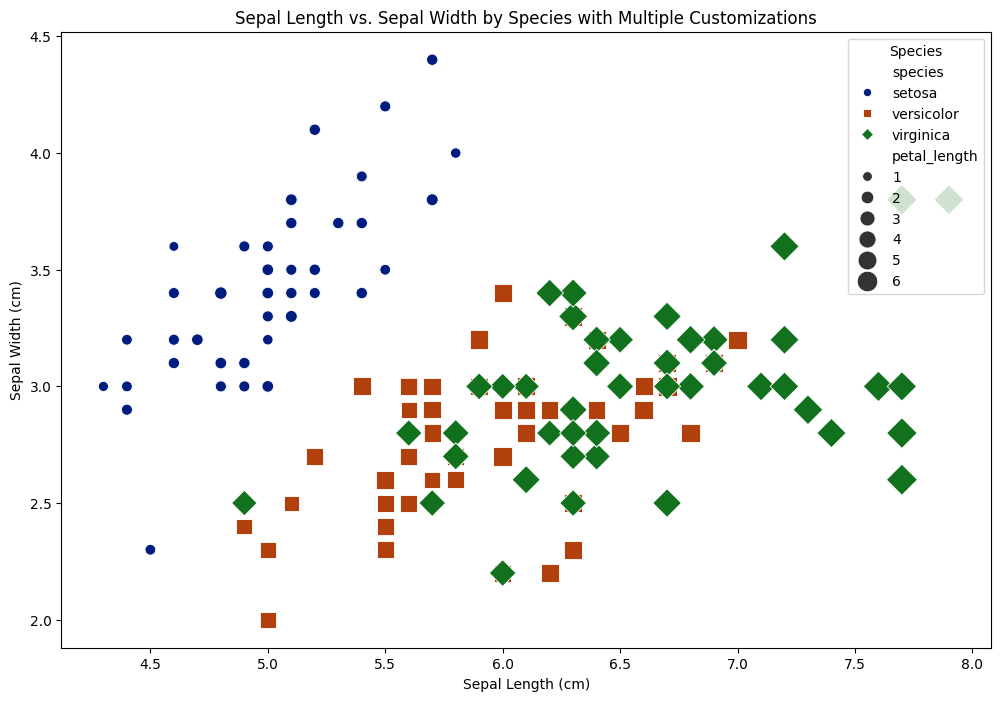
plt.title("Sepal Length vs. Sepal Width by Species with Multiple Customizations")

plt.xlabel("Sepal Length (cm)")

plt.ylabel("Sepal Width (cm)")

plt.legend(title="Species")

plt.show()



**Saving the Customized Plot**

Finally, you can save your customized plot using plt.savefig(). Saving the Customized Plot Description: This code snippet shows how to save the combined customized plot to a file.

Customizations:

Save Plot: The plot is saved as a PNG file named "customized\_scatter\_plot.png".

Figure Size: The figure size is set to 12x8 inches.

# Save the plot with customizations

plt.figure(figsize=(12, 8))

sns.scatterplot(

data=data,

x="sepal\_length",

y="sepal\_width",

hue="species",

style="species",

size="petal\_length",

sizes=(50, 250),

palette="dark",

markers=["o", "s", "D"]

)

plt.title("Sepal Length vs. Sepal Width by Species with Multiple Customizations")

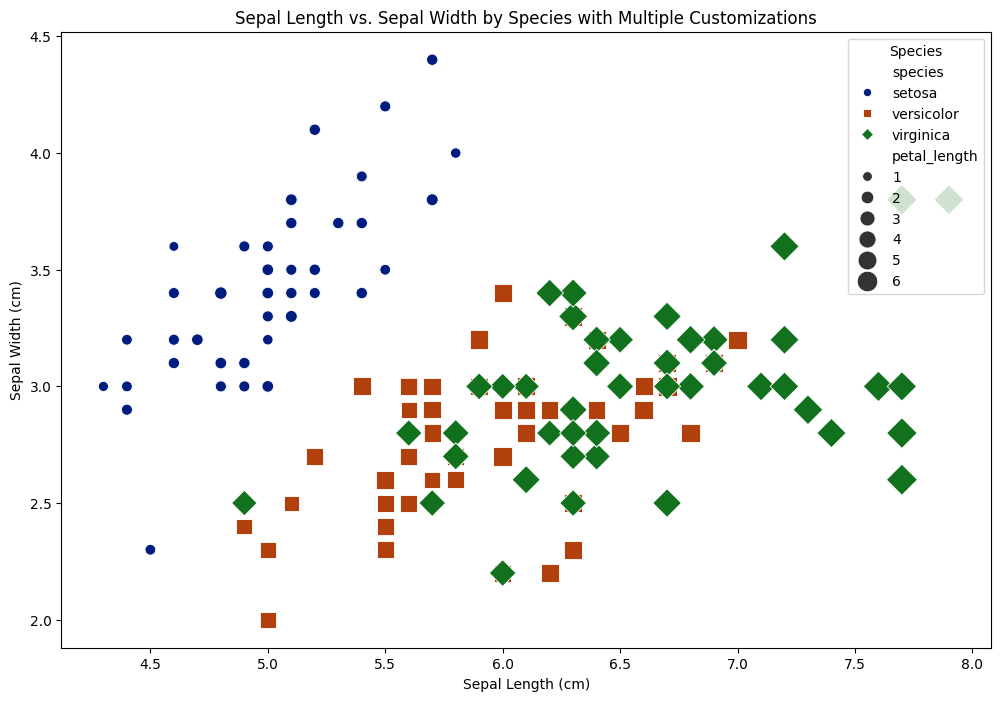
plt.xlabel("Sepal Length (cm)")

plt.ylabel("Sepal Width (cm)")

plt.legend(title="Species")

plt.savefig("customized\_scatter\_plot.png")

plt.show()



**SEABORN HEAT MAPS**

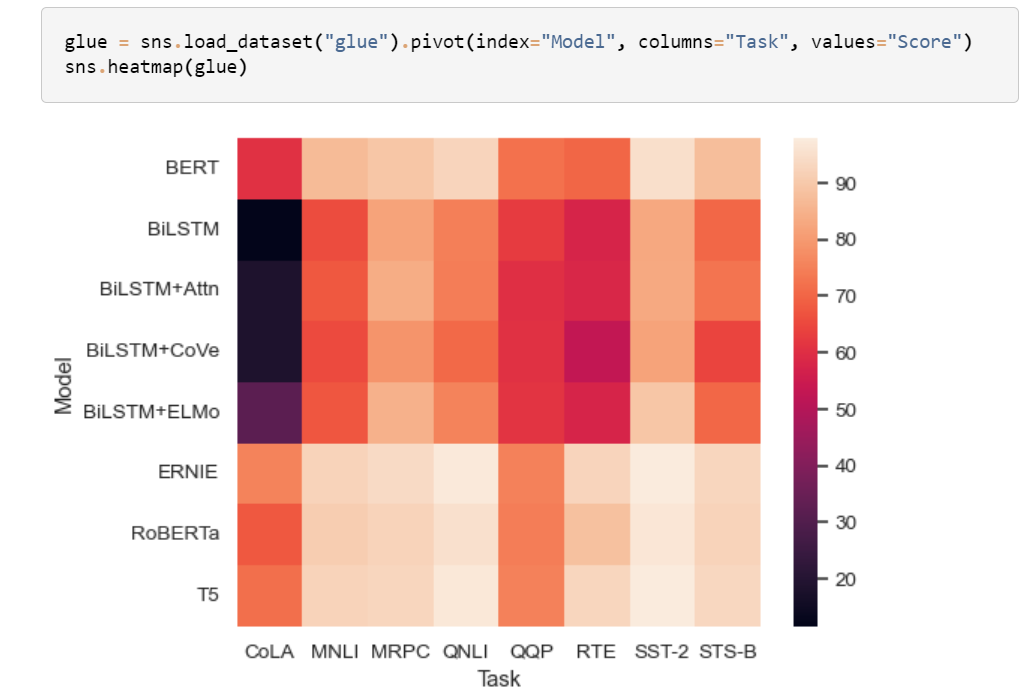
Plot rectangular data as a color-encoded matrix.

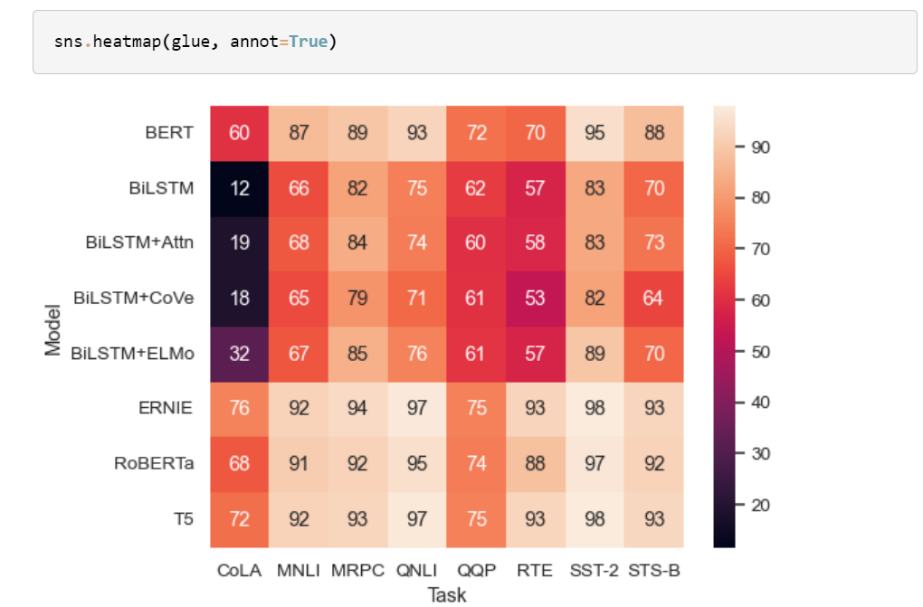
This is an Axes-level function and will draw the heatmap into the currently-active Axes if none is provided to the ax argument. Part of this Axes space will be taken and used to plot a colormap, unless cbar is False or a separate Axes is provided to cbar\_ax.

**Examples**

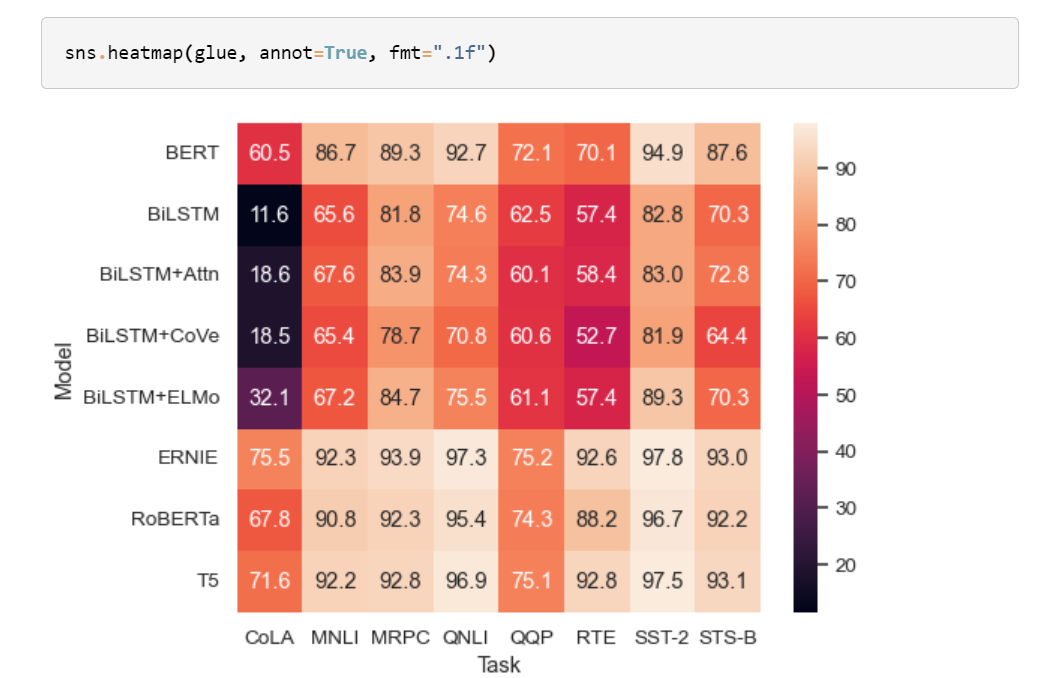
Pass a **DataFrame** to plot with indices as row/column labels:

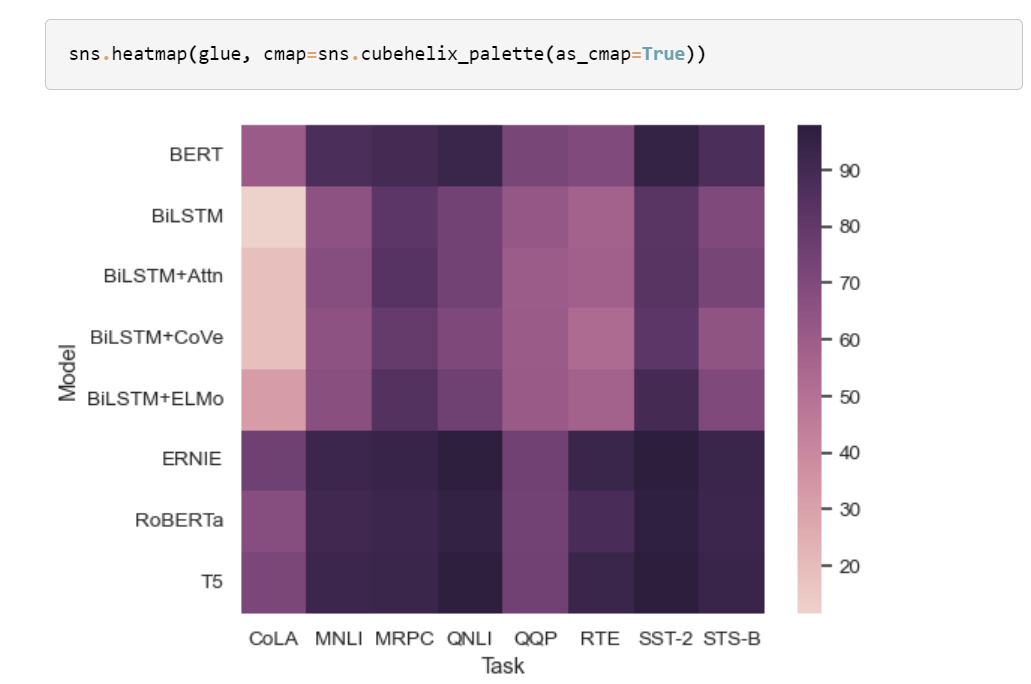
Use annot to represent the cell values with text:



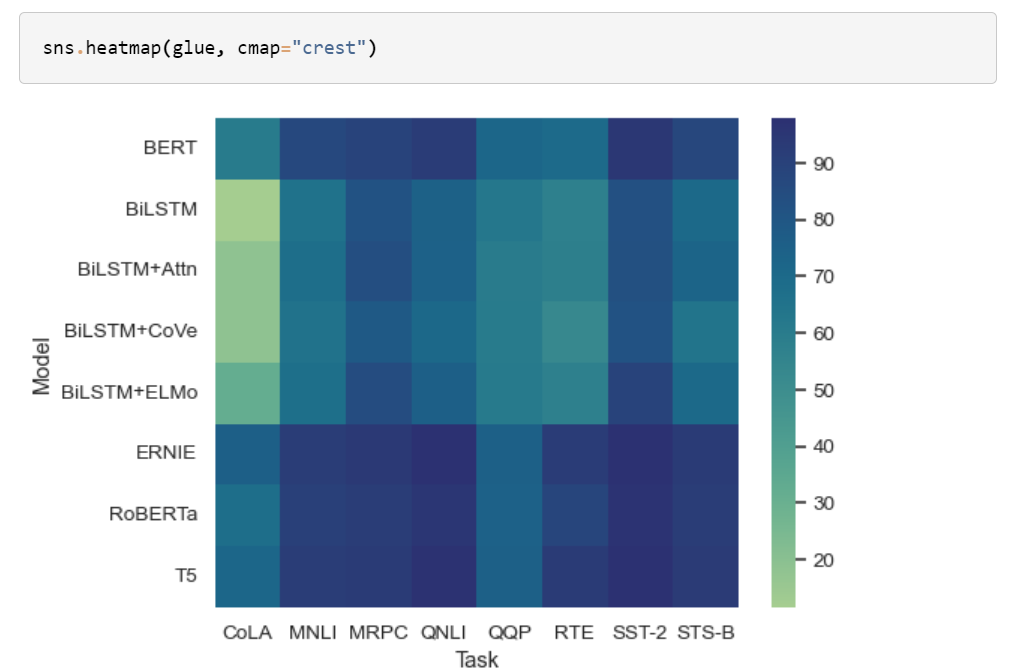
Control the annotations with a formatting string:

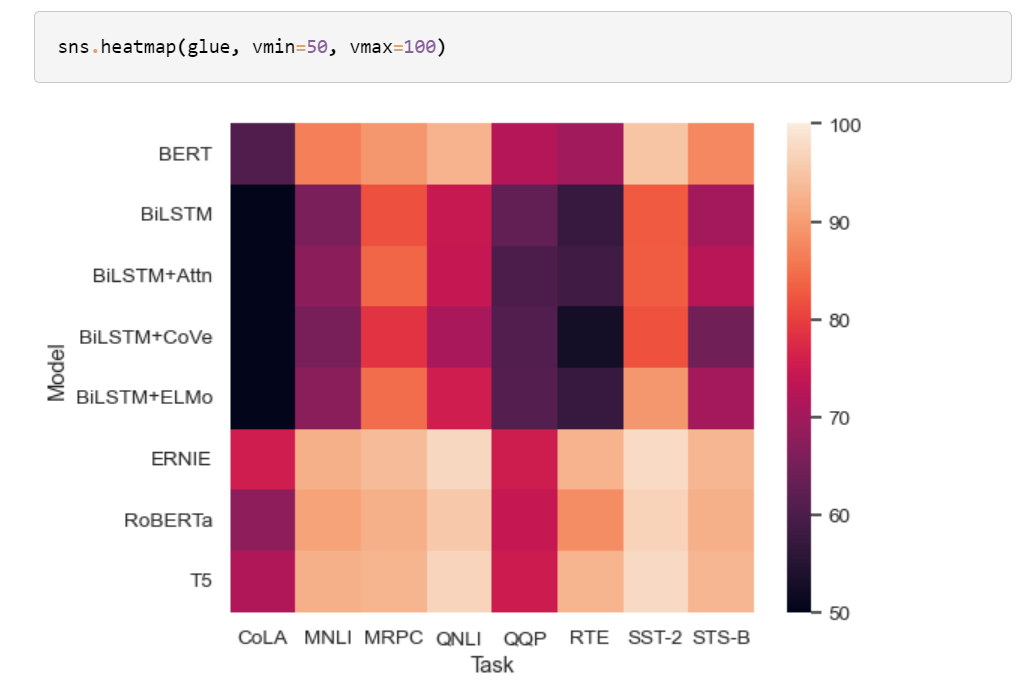
Use a separate dataframe for the annotations:



Selecting a different colormap by name:

Set the colormap norm (data values corresponding to minimum and maximum points):



Use methods on the [**matplotlib.axes.Axes**](https://matplotlib.org/stable/api/_as_gen/matplotlib.axes.Axes.html#matplotlib.axes.Axes) object to tweak the plot:

**SEABORN DISTRIBUTION PLOTS**

Figure-level interface for drawing distribution plots onto a Facet Grid.

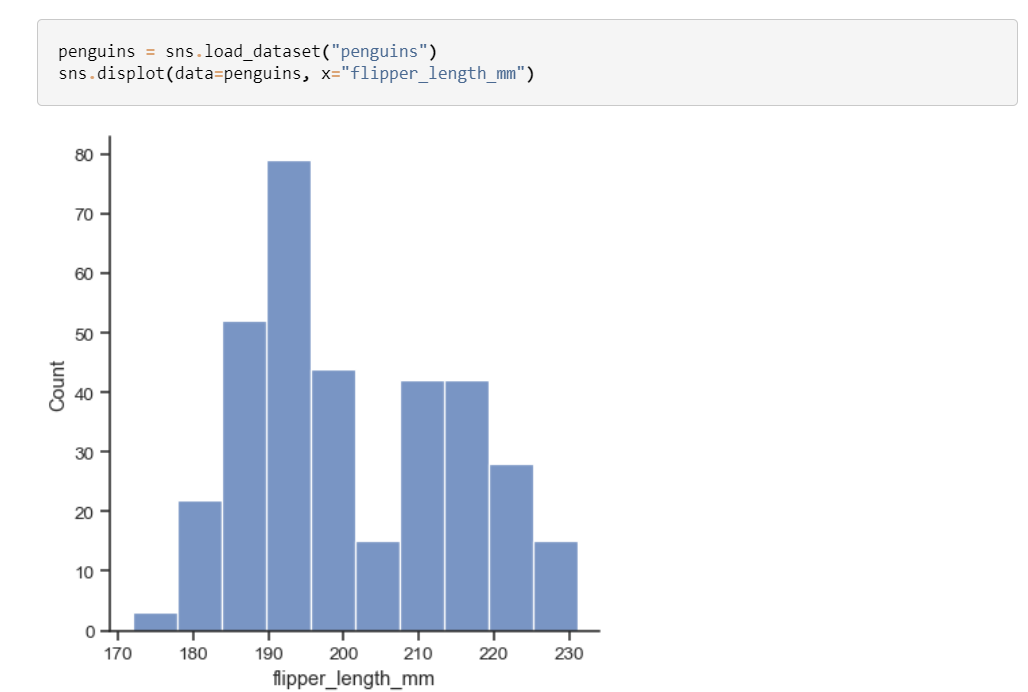
This function provides access to several approaches for visualizing the univariate or bivariate distribution of data, including subsets of data defined by semantic mapping and faceting across multiple subplots. The kind parameter selects the approach to use:

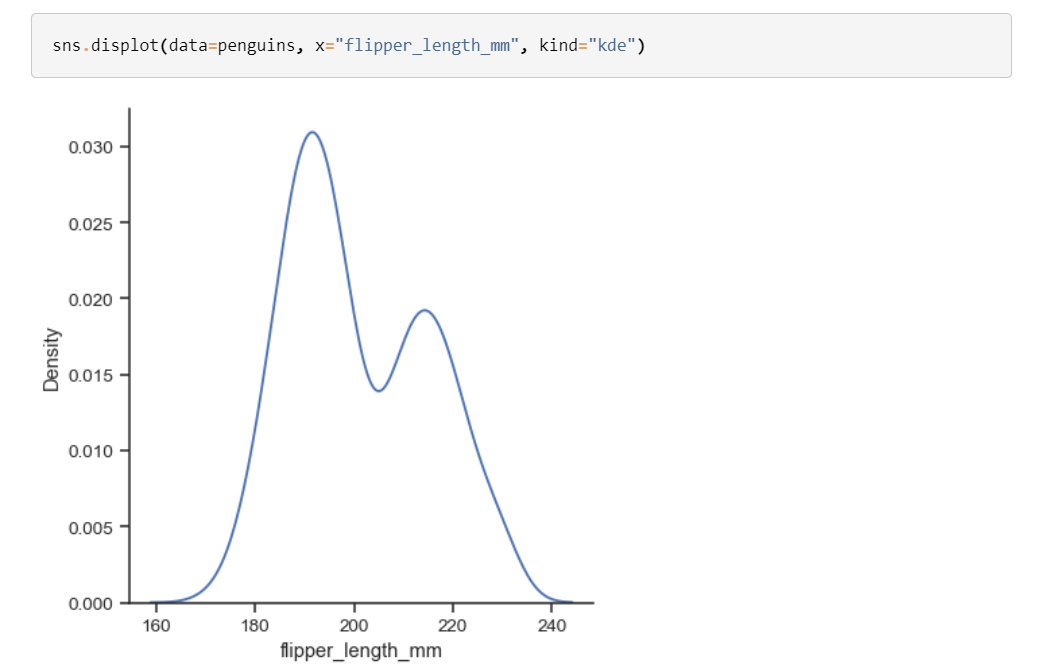
* [**histplot()**](https://seaborn.pydata.org/generated/seaborn.histplot.html#seaborn.histplot) (with kind="hist"; the default)
* [**kdeplot()**](https://seaborn.pydata.org/generated/seaborn.kdeplot.html#seaborn.kdeplot) (with kind="kde")
* [**ecdfplot()**](https://seaborn.pydata.org/generated/seaborn.ecdfplot.html#seaborn.ecdfplot) (with kind="ecdf"; univariate-only)

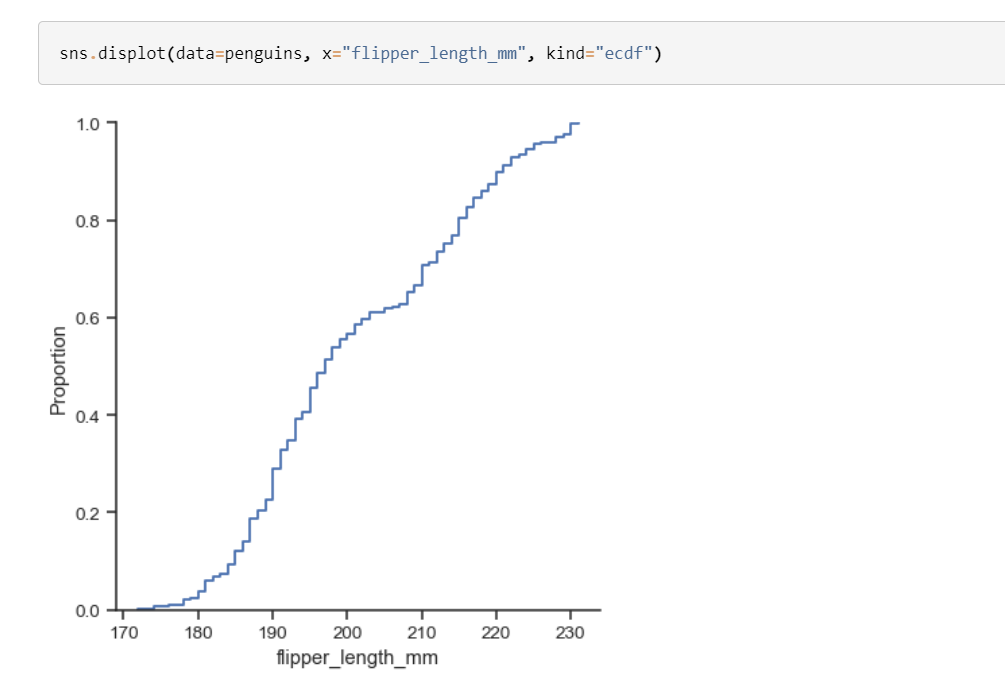
Additionally, a [**rugplot()**](https://seaborn.pydata.org/generated/seaborn.rugplot.html#seaborn.rugplot) can be added to any kind of plot to show individual observations.

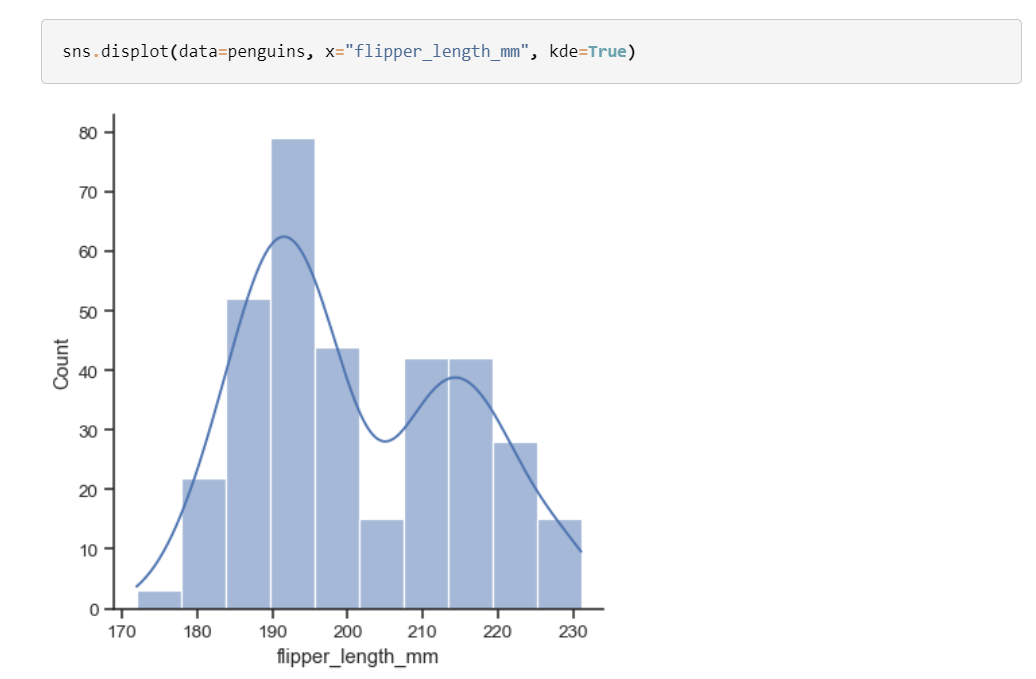
Extra keyword arguments are passed to the underlying function, so you should refer to the documentation for each to understand the complete set of options for making plots with this interface.

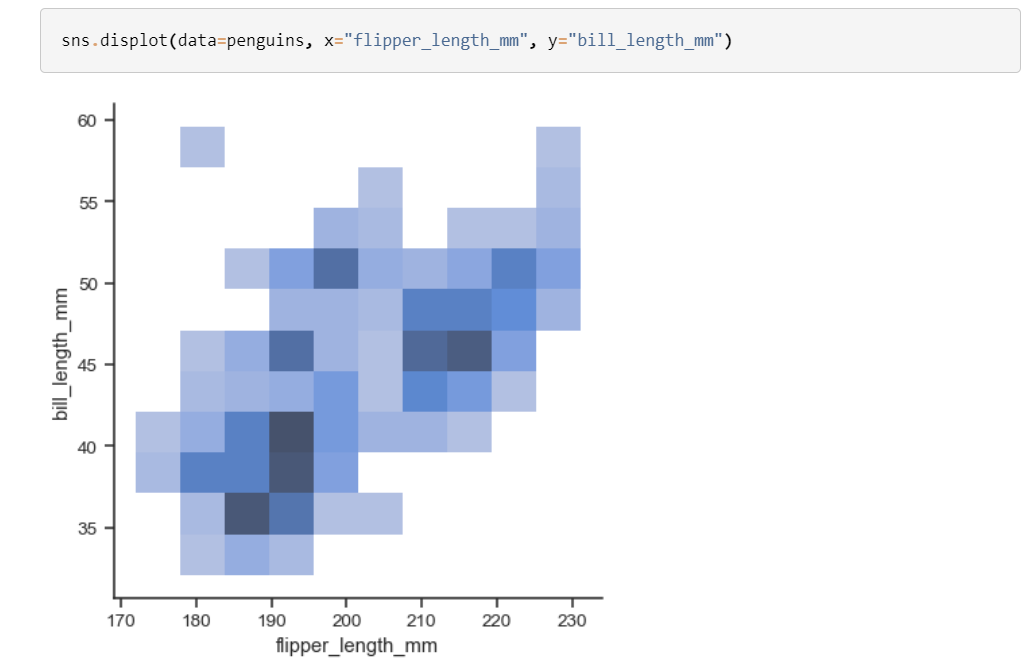
**Examples**

The default plot kind is a histogram:

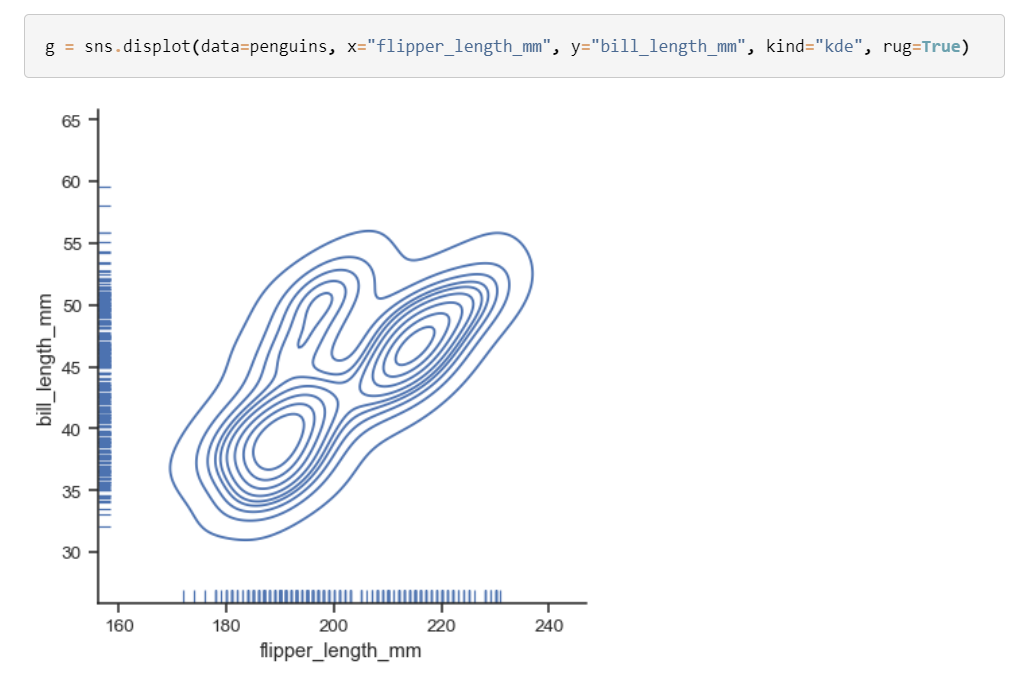
Use the kind parameter to select a different representation:

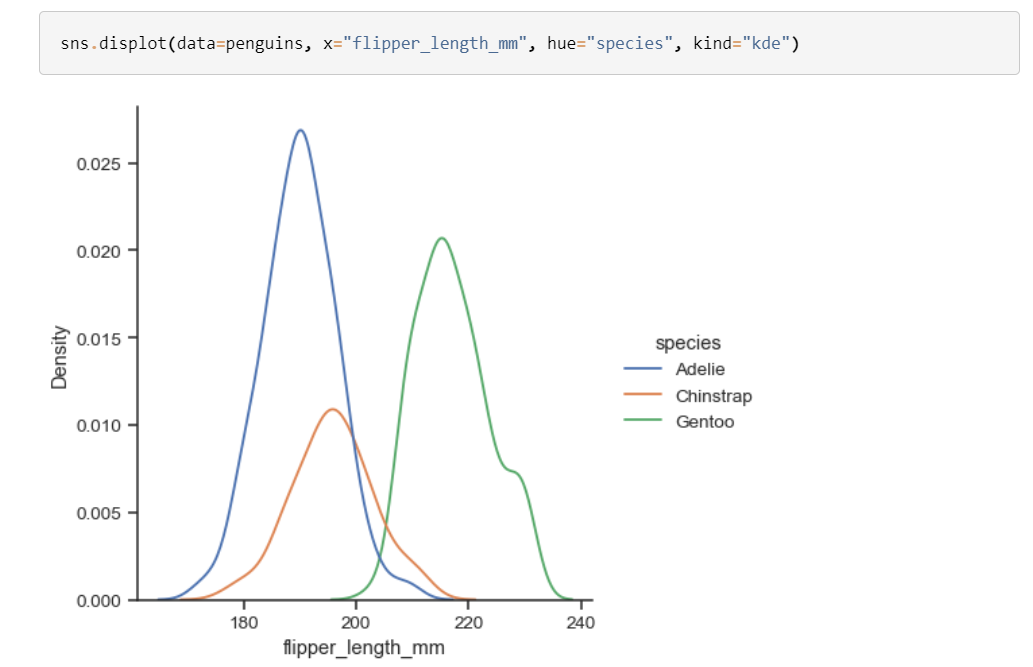
There are three main plot kinds; in addition to histograms and kernel density estimates (KDEs), you can also draw empirical cumulative distribution functions (ECDFs):While in histogram mode, it is also possible to add a KDE curve:

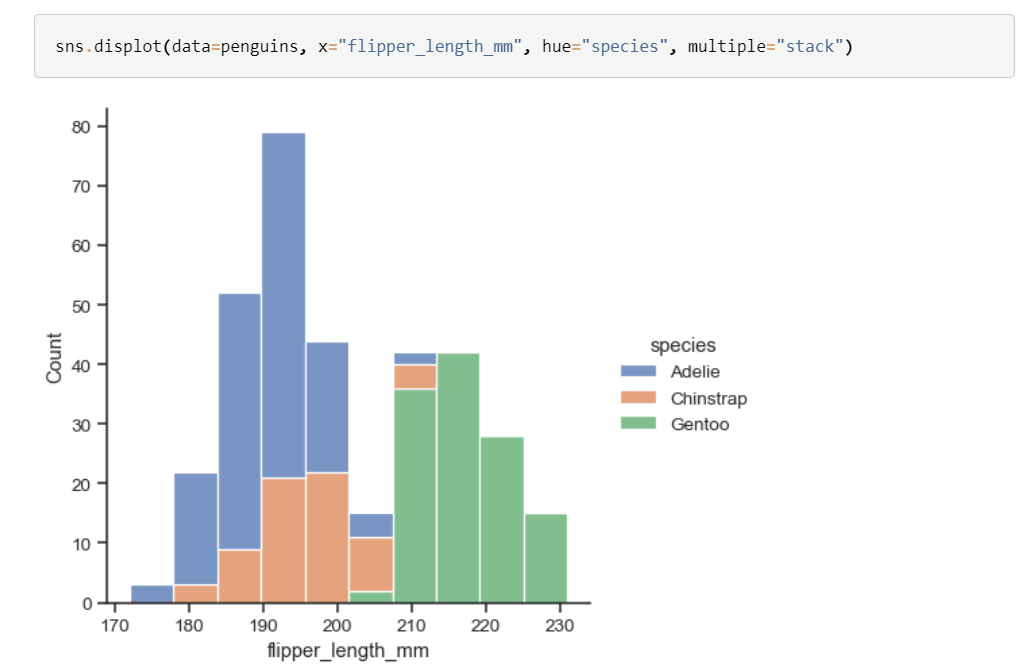
To draw a bivariate plot, assign both x and y:

Currently, bivariate plots are available only for histograms and KDEs:

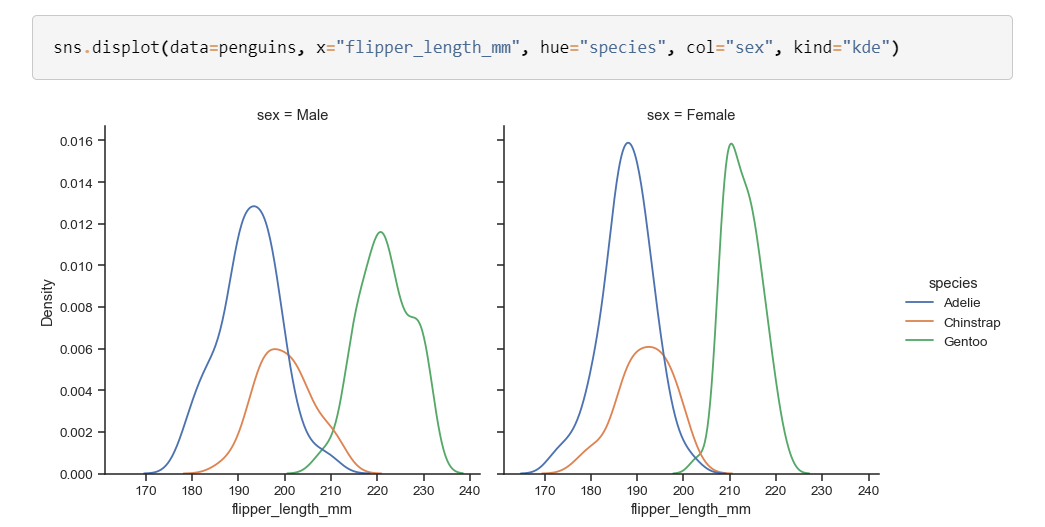
For each kind of plot, we can also show individual observations with a marginal “rug”:

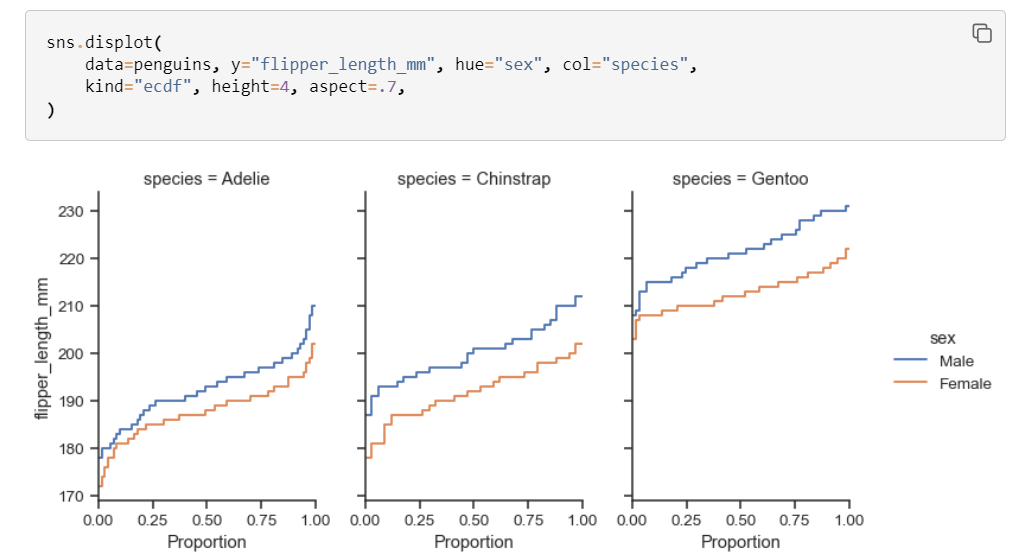
Each kind of plot can be drawn separately for subsets of data using hue mapping:

Additional keyword arguments are passed to the appropriate underlying plotting function, allowing for further customization:



The figure is constructed using a [**FacetGrid**](https://seaborn.pydata.org/generated/seaborn.FacetGrid.html#seaborn.FacetGrid), meaning that you can also show subsets on distinct subplots, or “facets”:

Because the figure is drawn with a [**FacetGrid**](https://seaborn.pydata.org/generated/seaborn.FacetGrid.html#seaborn.FacetGrid), you control its size and shape with the height and aspect parameters:



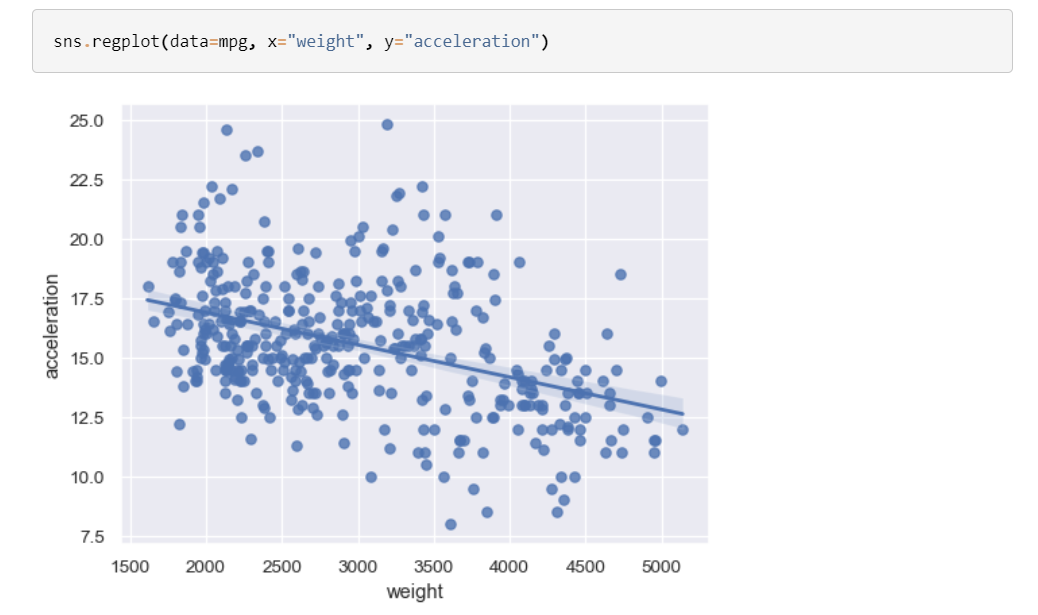
**SEABRON REGRESSION PLOTS**

Plot data and a linear regression model fit.

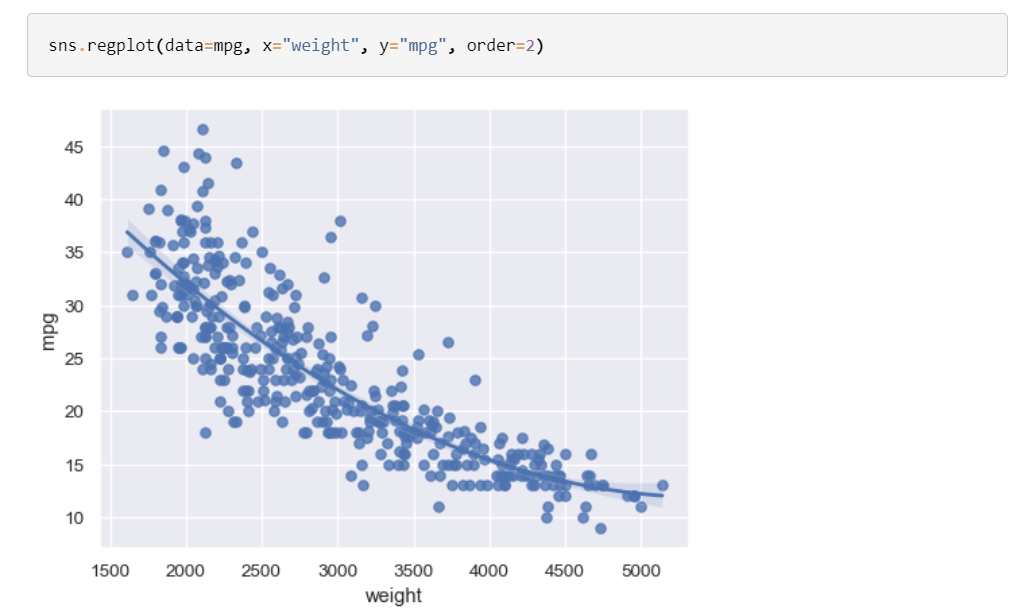
There are a number of mutually exclusive options for estimating the regression model. See the [tutorial](https://seaborn.pydata.org/tutorial/regression.html#regression-tutorial) for more information.

**Examples**

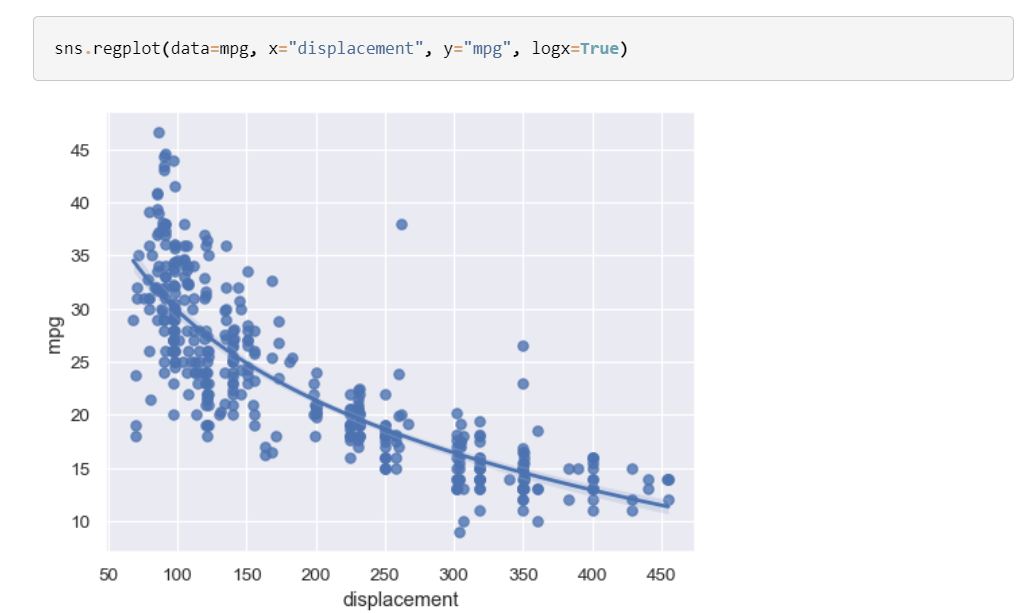
Plot the relationship between two variables in a DataFrame:



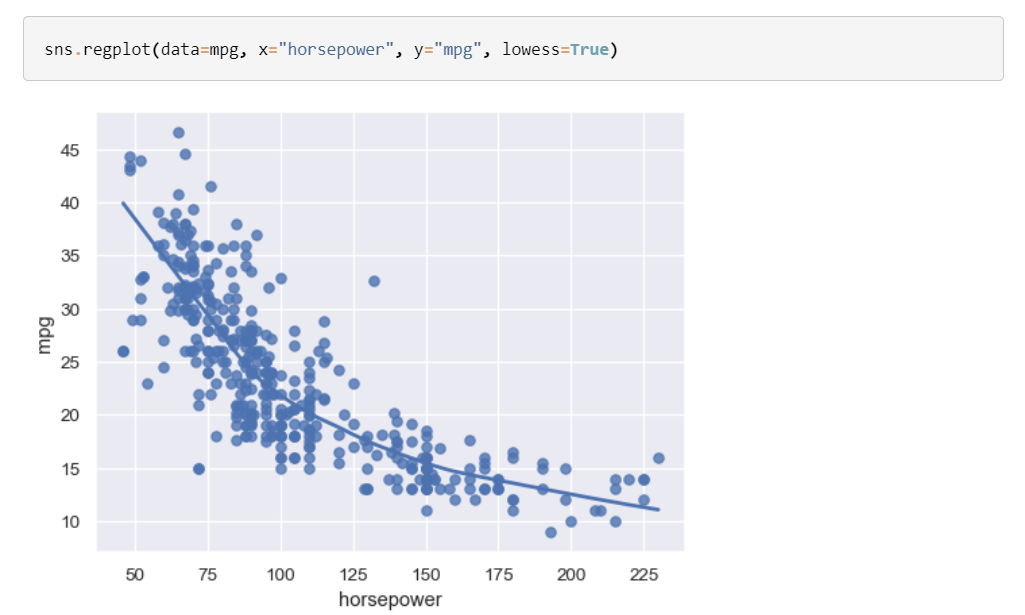
Fit a higher-order polynomial regression to capture nonlinear trends:



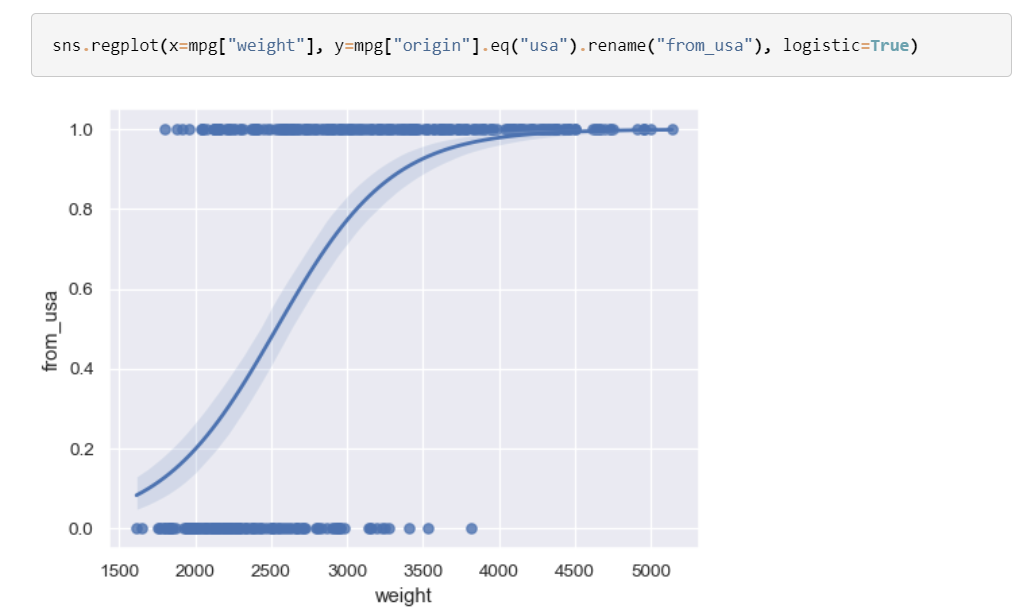
Alternatively, fit a log-linear regression:



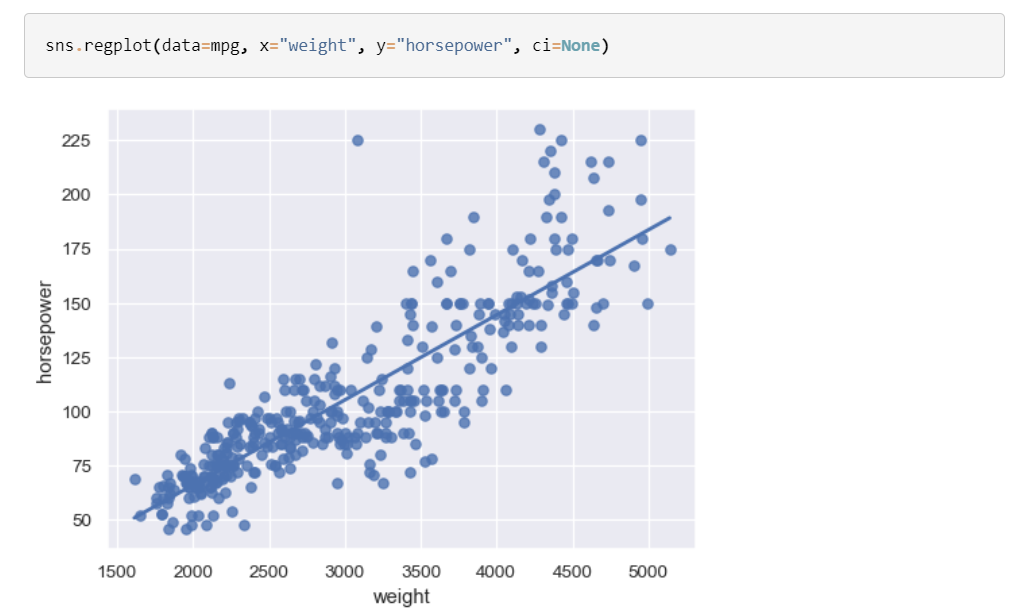
Or use a locally-weighted (LOWESS) smoother:



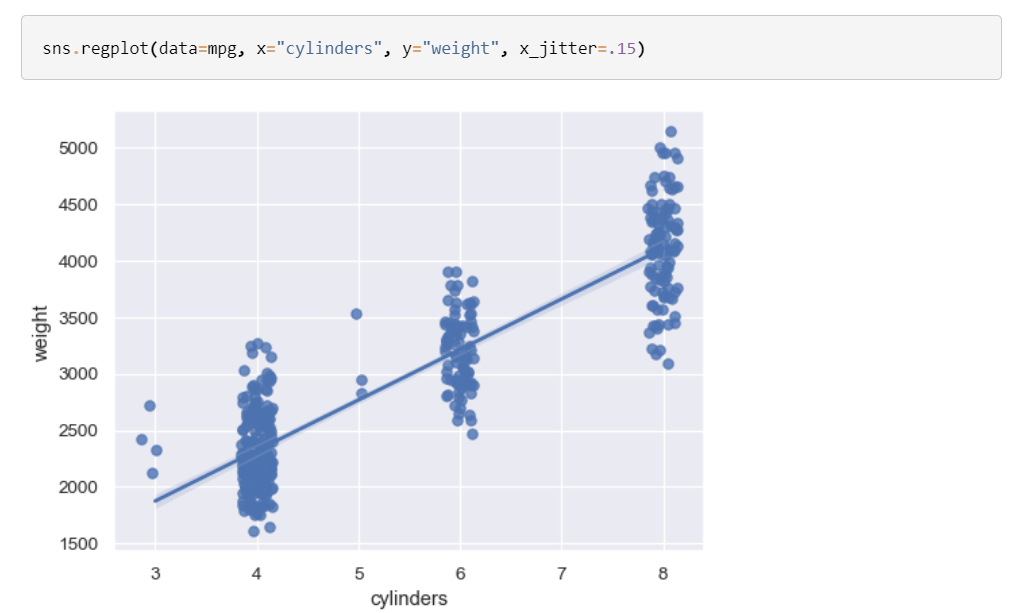
Fit a logistic regression when the response variable is binary:



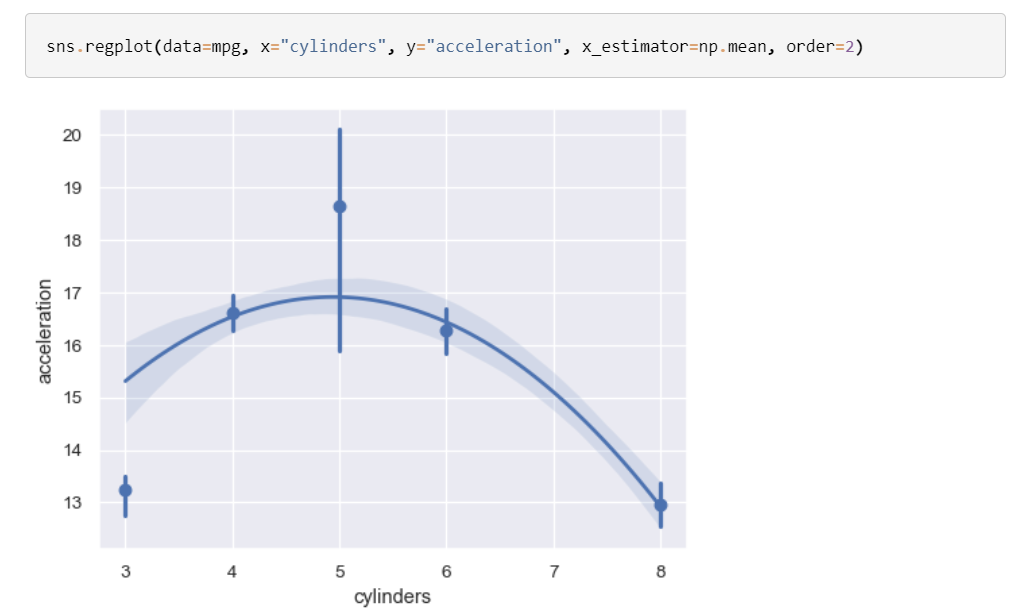
Disable the confidence interval for faster plotting:



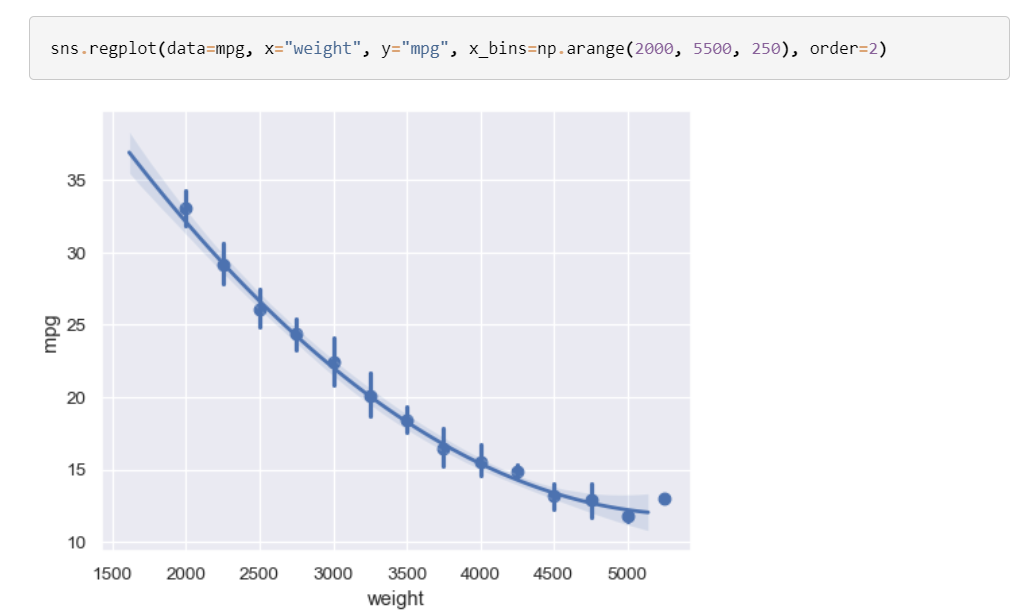
Jitter the scatterplot when the x variable is discrete:



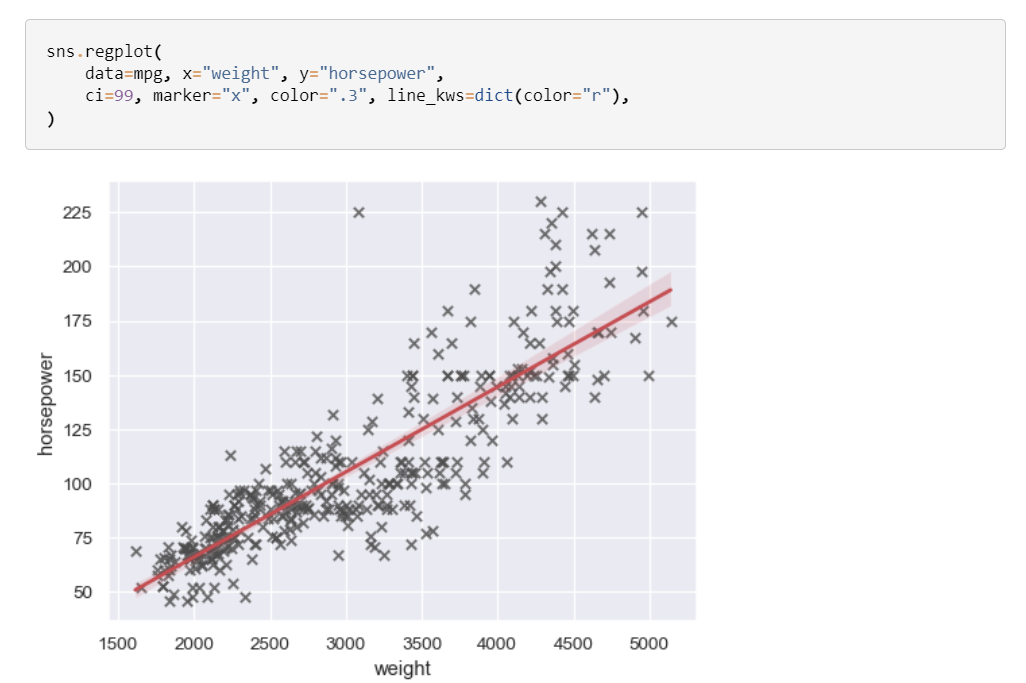
Or aggregate over the distinct x values:



With a continuous x variable, bin and then aggregate:



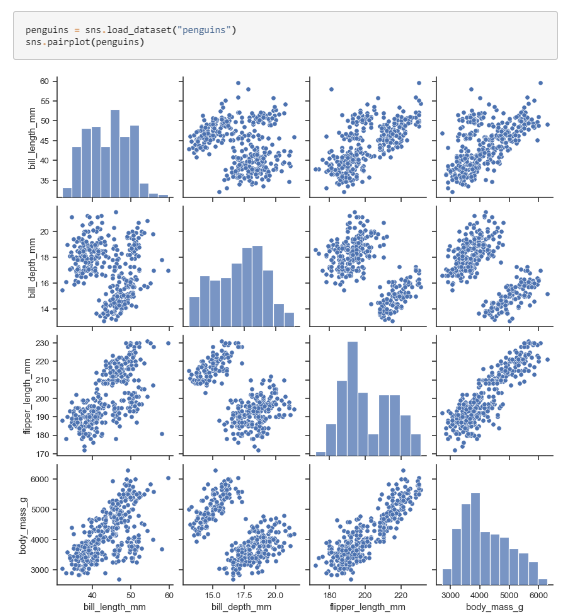
Customize the appearance of various elements:



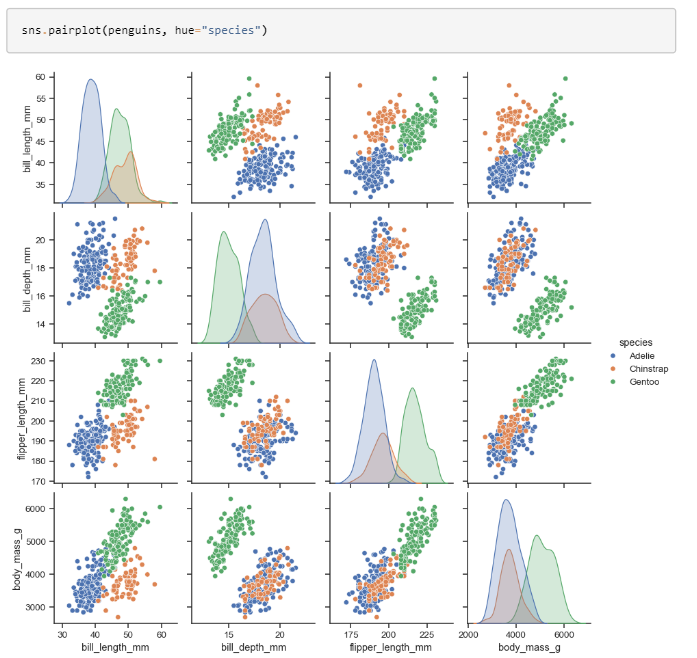
**SEABRON PAIRPLOTS**

**Examples**

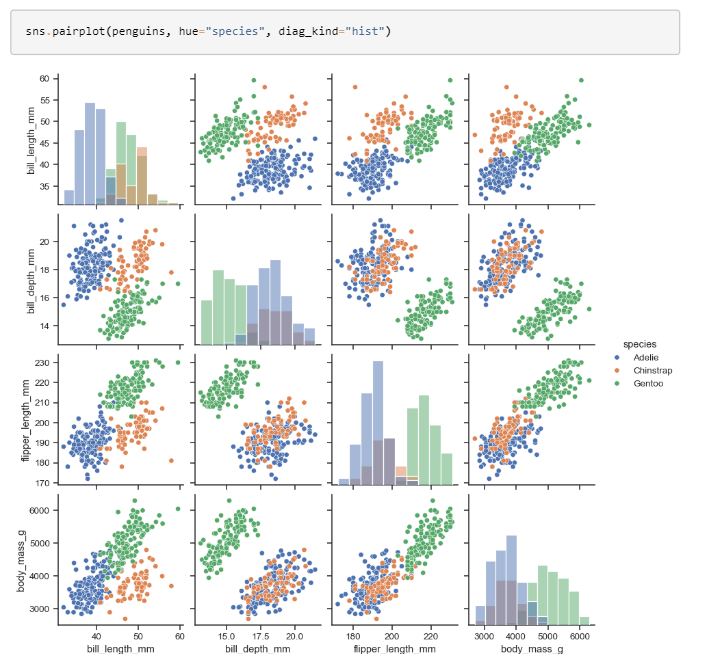
The simplest invocation uses [**scatterplot()**](https://seaborn.pydata.org/generated/seaborn.scatterplot.html#seaborn.scatterplot) for each pairing of the variables and [**histplot()**](https://seaborn.pydata.org/generated/seaborn.histplot.html#seaborn.histplot) for the marginal plots along the diagonal:



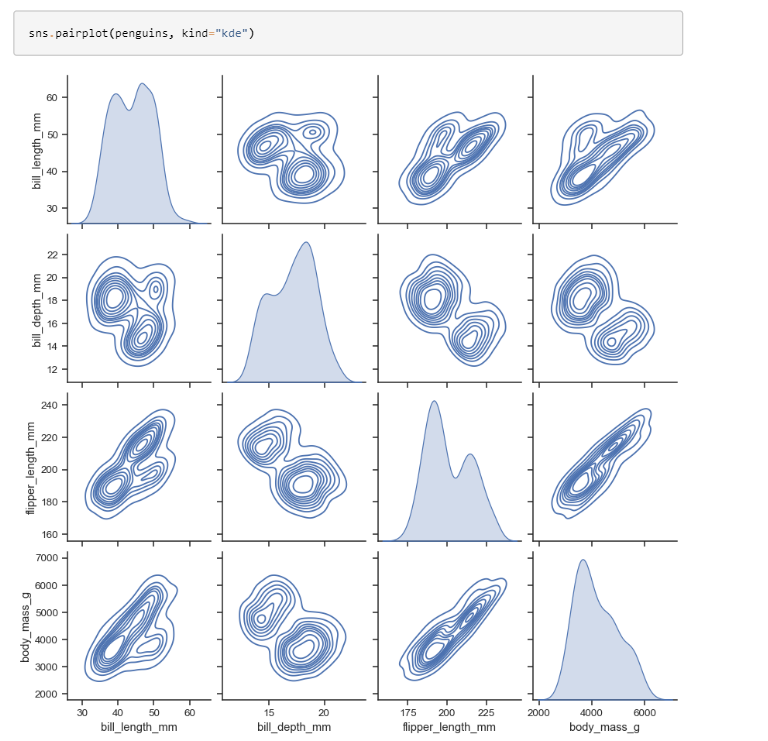
Assigning a hue variable adds a semantic mapping and changes the default marginal plot to a layered kernel density estimate (KDE):



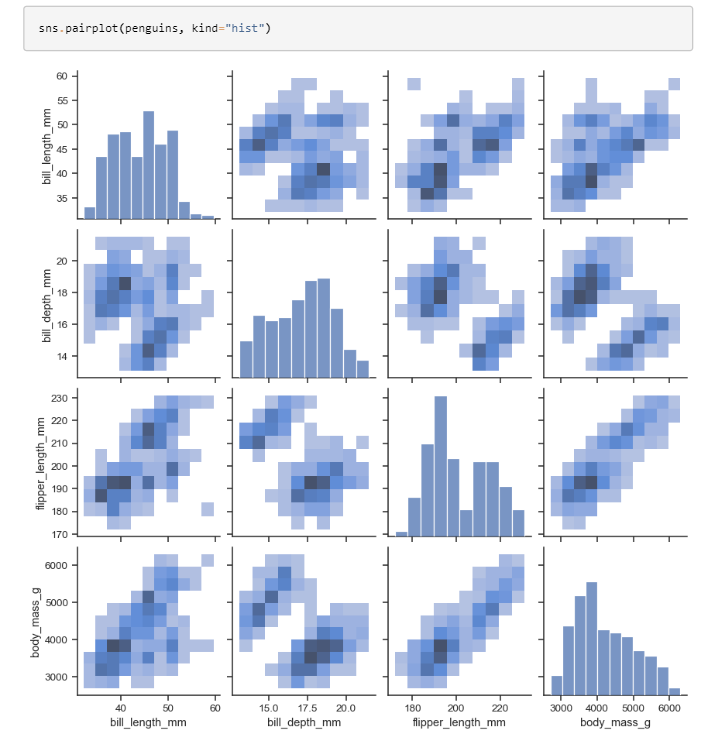
It’s possible to force marginal histograms:

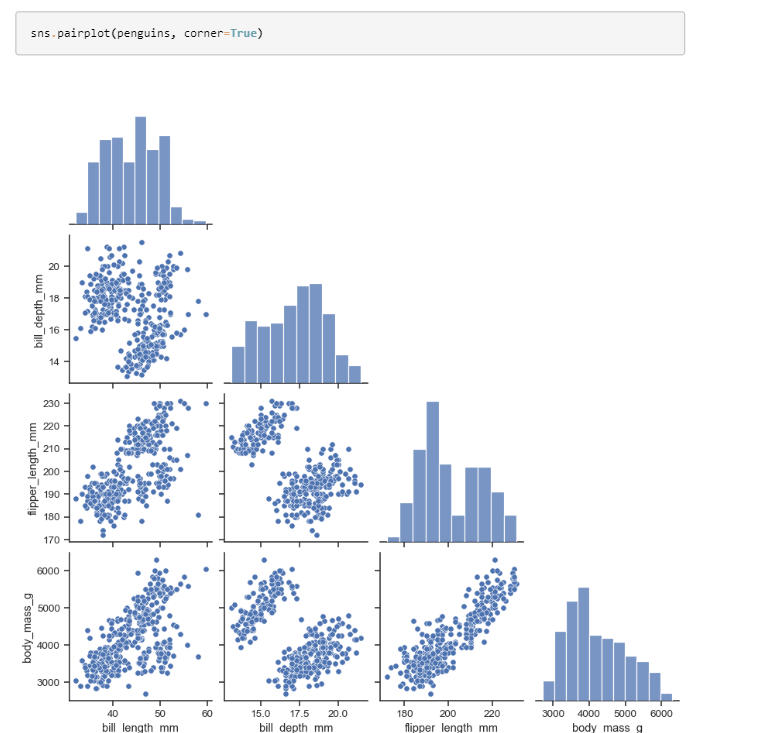


The kind parameter determines both the diagonal and off-diagonal plotting style. Several options are available, including using [**kdeplot()**](https://seaborn.pydata.org/generated/seaborn.kdeplot.html#seaborn.kdeplot) to draw KDEs:



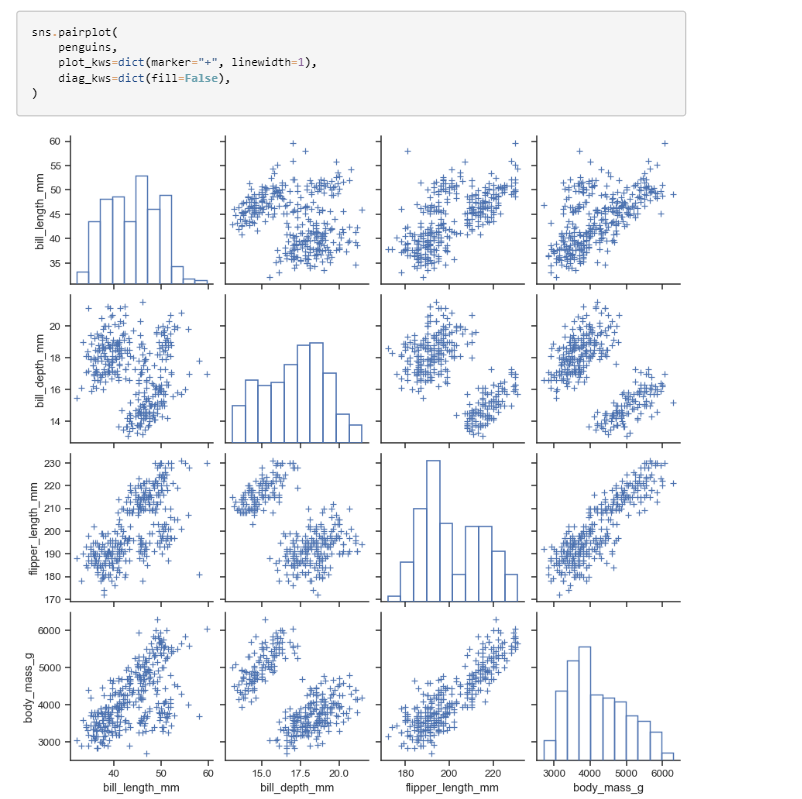
Or [**histplot()**](https://seaborn.pydata.org/generated/seaborn.histplot.html#seaborn.histplot) to draw both bivariate and univariate histograms:



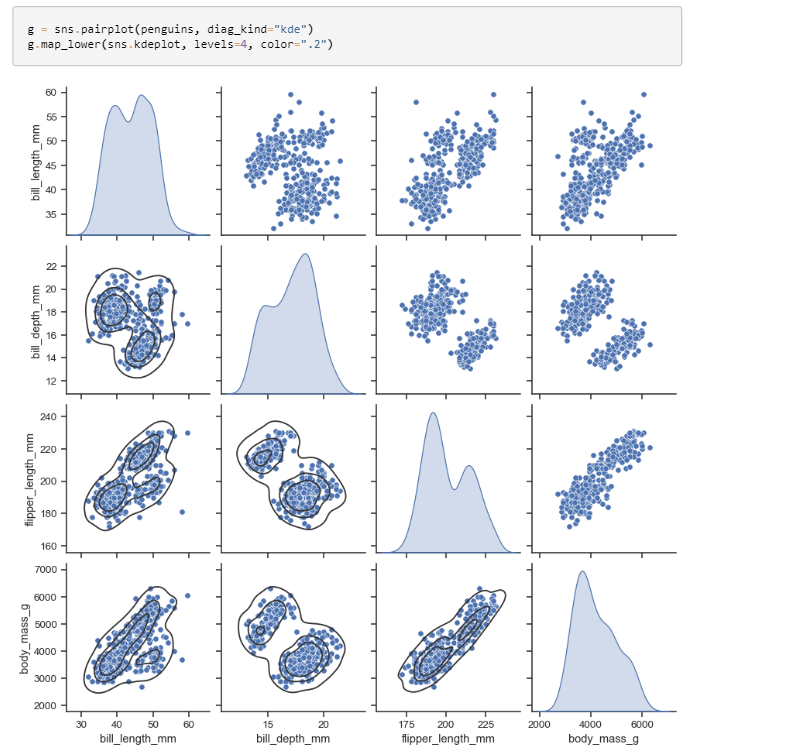
Use vars or x\_vars and y\_vars to select the variables to plot:

Set corner=True to plot only the lower triangle:

The plot\_kws and diag\_kws parameters accept dicts of keyword arguments to customize the off-diagonal and diagonal plots, respectively:



The return object is the underlying [**PairGrid**](https://seaborn.pydata.org/generated/seaborn.PairGrid.html#seaborn.PairGrid), which can be used to further customize the plot:



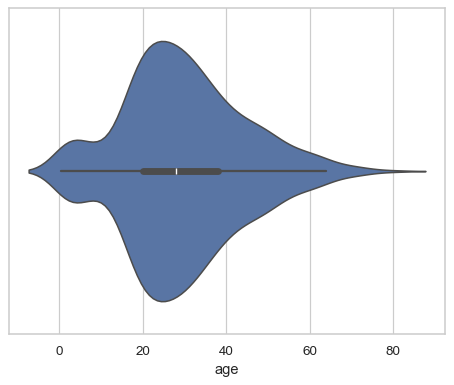
**SEABORN VIOLIN PLOTS**

**Examples :**

The default violinplot represents a distribution two ways: a patch showing a symmetric kernel density estimate (KDE), and the quartiles / whiskers of a box plot:

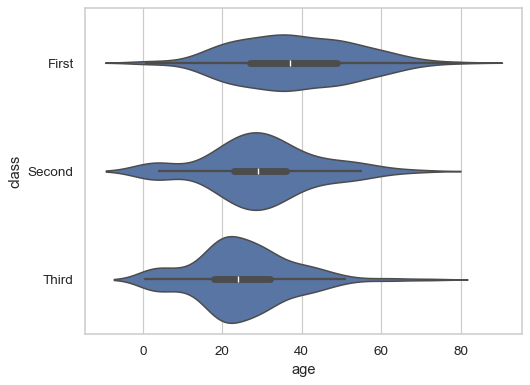
df **=** sns**.**load\_dataset**(**"titanic"**)**

sns**.**violinplot**(**x**=**df**[**"age"**])**



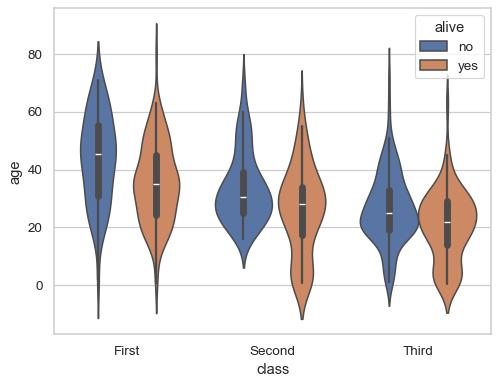
In a bivariate plot, one of the variables will “group” so that multiple violins are drawn:

sns**.**violinplot**(**data**=**df**,** x**=**"age"**,** y**=**"class"**)**



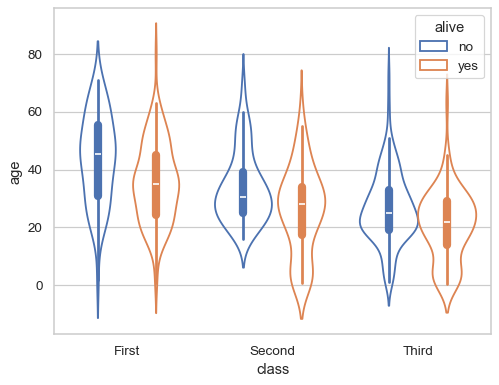
By default, the orientation of the plot is determined by the variable types, preferring to group by a categorical variable:

sns**.**violinplot**(**data**=**df**,** x**=**"class"**,** y**=**"age"**,** hue**=**"alive"**)**



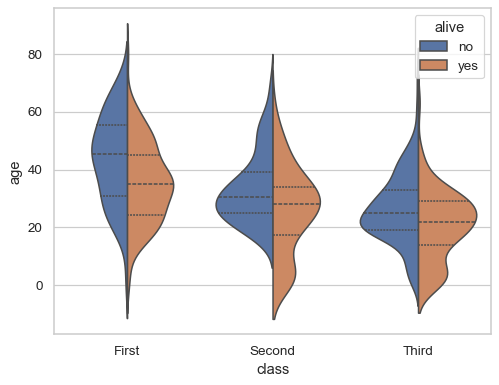
Pass fill=False to draw line-art violins:

sns**.**violinplot**(**data**=**df**,** x**=**"class"**,** y**=**"age"**,** hue**=**"alive"**,** fill**=False)**



Draw “split” violins to take up less space, and only show the data quarties:

sns**.**violinplot**(**data**=**df**,** x**=**"class"**,** y**=**"age"**,** hue**=**"alive"**,** split**=True,** inner**=**"quart"**)**



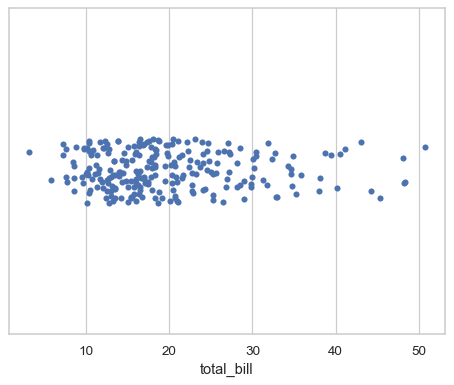
**SEABRON STRIPPLOTS**

**Examples**

Assigning a single numeric variable shows its univariate distribution with points randomly “jittered” on the other axis:

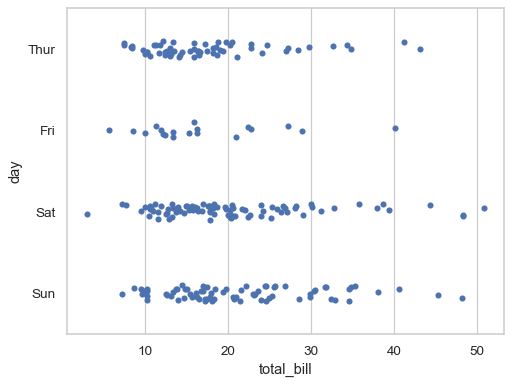
tips **=** sns**.**load\_dataset**(**"tips"**)**

sns**.**stripplot**(**data**=**tips**,** x**=**"total\_bill"**)**



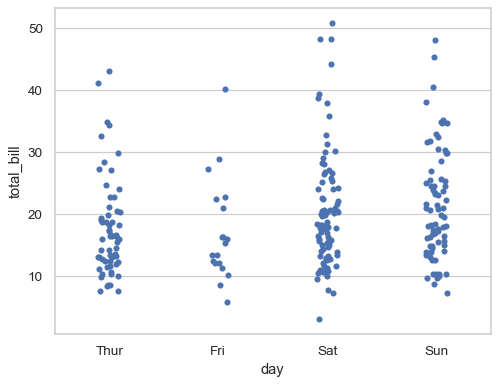
Assigning a second variable splits the strips of points to compare categorical levels of that variable:

sns**.**stripplot**(**data**=**tips**,** x**=**"total\_bill"**,** y**=**"day"**)**



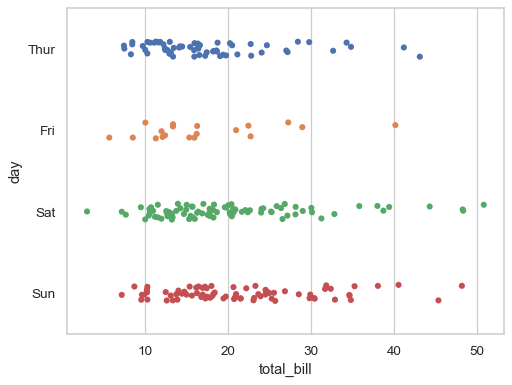
Show vertically-oriented strips by swapping the assignment of the categorical and numerical variables:

sns**.**stripplot**(**data**=**tips**,** x**=**"day"**,** y**=**"total\_bill"**)**



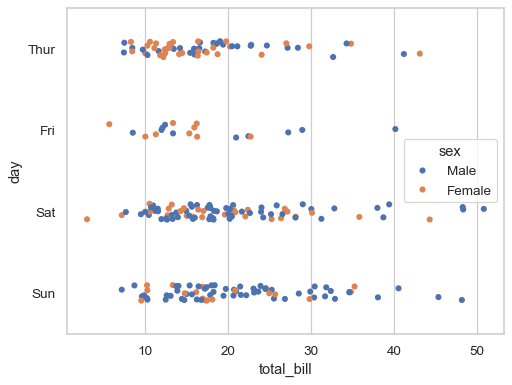
Prior to version 0.12, the levels of the categorical variable had different colors by default. To get the same effect, assign the hue variable explicitly:

sns**.**stripplot**(**data**=**tips**,** x**=**"total\_bill"**,** y**=**"day"**,** hue**=**"day"**,** legend**=False)**



Or you can assign a distinct variable to hue to show a multidimensional relationship:

sns**.**stripplot**(**data**=**tips**,** x**=**"total\_bill"**,** y**=**"day"**,** hue**=**"sex"**)**

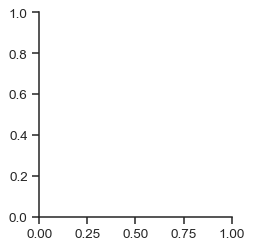


**SEABORN FACETGRIDS**

**Examples**

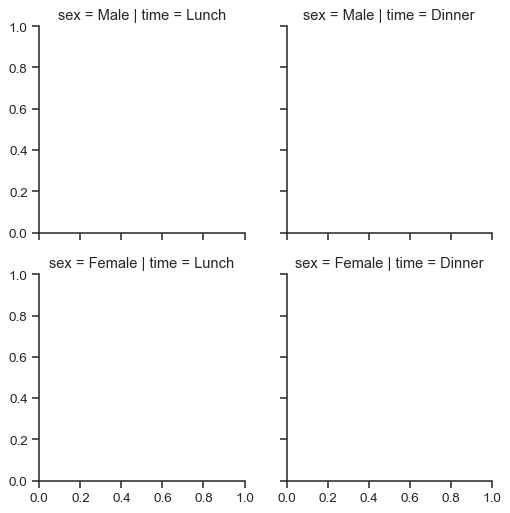
Calling the constructor requires a long-form data object. This initializes the grid, but doesn’t plot anything on it:

tips **=** sns**.**load\_dataset**(**"tips"**)**sns**.**FacetGrid**(**tips**)**

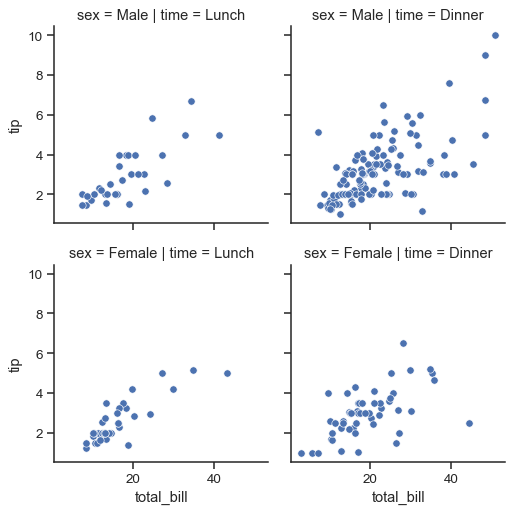


Assign column and/or row variables to add more subplots to the figure:

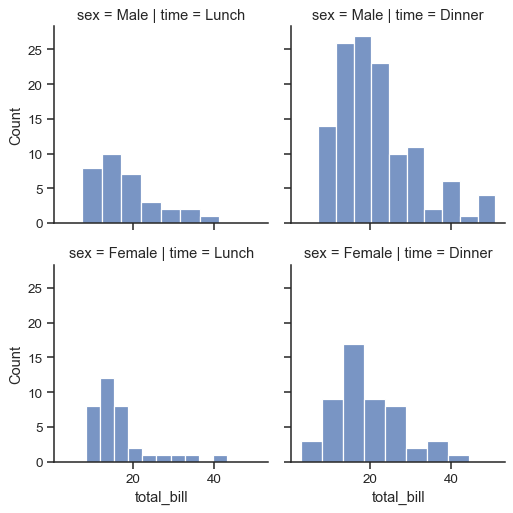
sns**.**FacetGrid**(**tips**,** col**=**"time"**,** row**=**"sex"**)**



To draw a plot on every facet, pass a function and the name of one or more columns in the dataframe to **[FacetGrid.map()](https://seaborn.pydata.org/generated/seaborn.FacetGrid.map.html" \l "seaborn.FacetGrid.map" \o "seaborn.FacetGrid.map)**:



The variable specification in **[FacetGrid.map()](https://seaborn.pydata.org/generated/seaborn.FacetGrid.map.html" \l "seaborn.FacetGrid.map" \o "seaborn.FacetGrid.map)** requires a positional argument mapping, but if the function has a data parameter and accepts named variable assignments, you can also use **[FacetGrid.map\_dataframe()](https://seaborn.pydata.org/generated/seaborn.FacetGrid.map_dataframe.html" \l "seaborn.FacetGrid.map_dataframe" \o "seaborn.FacetGrid.map_dataframe)**:



Notice how the bins have different widths in each facet. A separate plot is drawn on each facet, so if the plotting function derives any parameters from the data, they may not be shared across facets. You can pass additional keyword arguments to synchronize them. But when possible, using a figure-level function like [**displot()**](https://seaborn.pydata.org/generated/seaborn.displot.html#seaborn.displot) will take care of this bookkeeping for you:



**BEST PRACTICES FOR DATA VISUALIZATION**

**WITH SEABORN**

**1. Understanding and Preparing Your Data**

Before visualizing your data, it is crucial to thoroughly understand and prepare it:

**1.Data Inspection**: Familiarize yourself with the structure, types, and distributions within your dataset. Use pandas functions like .info(), .describe(), and .head() to get a preliminary overview.

import pandas as pd

df = pd.read\_csv("data.csv")

print(df.info())

print(df.describe())

print(df.head())

**Data Cleaning**: Address missing values, remove duplicates, and handle outliers to ensure data integrity. Techniques such as imputation or dropping rows/columns can be employed based on the context.

**2.Choosing the Right Plot**

Selecting the appropriate plot type is fundamental to conveying the correct message:

**Categorical Data**: Visualize categorical variables using bar plots, count plots, or box plots

sns.barplot(x="category", y="value", data=df)

**Continuous Data**: For continuous variables, use histograms, KDE plots, or scatter plots.

sns.histplot(df["value"], bins=30)

**Comparisons**: Line plots are ideal for showing trends over time

sns.lineplot(x="date", y="value", data=df)

**Correlations**: Heatmaps are effective for displaying the correlation matrix

sns.heatmap(df.corr(), annot=True, cmap="coolwarm")

**3. Customizing for Clarity**

Enhancing the clarity of your visualizations ensures that your audience can easily interpret them:

* **Labels and Titles**: Always label your axes and provide a title for context. Use clear, descriptive titles and axis labels.

ax = sns.barplot(x="category", y="value", data=df)

ax.set\_title("Value by Category")

ax.set\_xlabel("Category")

ax.set\_ylabel("Value")

plt.show()

**4. Aesthetics**

Using Seaborn’s aesthetic settings can significantly enhance the readability and visual appeal of your plots:

**Themes**: Utilize Seaborn's built-in themes for consistent and attractive visual styles.

sns.set\_theme(style="darkgrid")

**Color Palettes**: Choose appropriate color palettes that are visually appealing and accessible to colorblind users.

sns.set\_palette("colorblind")

**5. Faceting**

Faceting is a powerful technique to visualize subsets of your data:

**FacetGrid**: Use FacetGrid to create a grid of plots based on subsets of your data. This is particularly useful for comparing distributions across different categories.

g = sns.FacetGrid(df, col="category", row="sub\_category")

g.map(sns.histplot, "value")

**6. Annotations**

Annotations can highlight significant data points or trends, adding depth to your analysis:

**Adding Annotations**: Use annotations to emphasize important points or outliers in your plots

ax = sns.scatterplot(x="total\_bill", y="tip", data=df)

ax.annotate('Outlier', xy=(50, 10), xytext=(55, 12),

arrowprops=dict(facecolor='black', shrink=0.05))

plt.show()

**7. Maintaining Aspect Ratio**

Proper aspect ratio ensures that your visualizations are not distorted and remain readable:

**Setting Figure Size**: Adjust the figure size to maintain an appropriate aspect ratio

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

**8. Enhancing Interactivity**

For more complex visualizations, consider using interactive plots:

**Interactive Plots**: Combine Seaborn with Plotly to add interactivity, allowing users to explore the data dynamically.

import plotly.express as px

fig = px.scatter(df, x="total\_bill", y="tip", color="day")

fig.show()

**9. Ensuring Reproducibility**

Reproducibility is key in data science to ensure that results can be verified and built upon:

**Code Readability**: Write clear, well-commented code to make it understandable and maintainable.

**Version Control**: Use version control systems like Git to track changes in your code and data visualizations.

By following these best practices, you can leverage Seaborn to create effective, insightful, and visually appealing data visualizations that communicate your findings clearly and convincingly.

**Conclusion:-**

### Effective Data Visualization with Seaborn

In the realm of data analysis, visualization plays a pivotal role in understanding and interpreting data. Seaborn, a powerful Python data visualization library built on top of Matplotlib, provides an extensive suite of tools to create informative and attractive statistical graphics. By adhering to best practices, users can maximize the potential of Seaborn to convey complex data insights in a clear and compelling manner.

**1. Understanding and Preparing Data:** Effective visualization begins with a thorough understanding of the data. Inspecting, cleaning, and preparing the dataset ensures that the visualizations are accurate and meaningful. This step involves handling missing values, removing duplicates, and addressing outliers to maintain data integrity.

**2. Choosing the Right Plot:** Selecting the appropriate type of plot for the data is crucial. Whether dealing with categorical or continuous data, or comparing trends and correlations, choosing the correct plot type—such as bar plots, histograms, line plots, or heatmaps—helps in accurately representing the data.

**3. Customizing for Clarity:** Customization enhances the readability of visualizations. Adding clear labels, titles, and legends makes it easier for the audience to understand the context and content of the plots. Thoughtful customization ensures that the key messages are not lost in the visualization.

**4. Aesthetics:** Using Seaborn’s themes and color palettes improves the visual appeal and readability of plots. Themes provide a consistent style, while color palettes ensure that the visualizations are accessible, even to those with color vision deficiencies.

**5. Faceting:** Faceting allows for the visualization of data subsets, making it easier to compare different categories or subgroups within the dataset. The FacetGrid functionality in Seaborn is particularly useful for creating grids of plots based on these subsets.

**6. Annotations:** Annotations highlight significant data points or trends within the visualization, adding depth and context to the analysis. By drawing attention to key areas, annotations help in better understanding and interpreting the data.

**7. Maintaining Aspect Ratio:** Maintaining an appropriate aspect ratio ensures that visualizations are not distorted and remain clear and readable. Adjusting the figure size appropriately helps in achieving this goal.

**8. Enhancing Interactivity:** For more detailed analysis, incorporating interactivity into visualizations can be highly beneficial. Combining Seaborn with libraries like Plotly enables users to create interactive plots that allow for dynamic exploration of the data.

**9. Ensuring Reproducibility:** Reproducibility is a cornerstone of data science. Writing clear, well-commented code and using version control systems ensure that visualizations can be easily reproduced, verified, and built upon by others.

### Summary

By following these best practices, data analysts and scientists can leverage Seaborn to its full potential, creating visualizations that are not only informative but also aesthetically pleasing and easy to understand. Seaborn’s capabilities, combined with careful preparation, thoughtful customization, and a focus on clarity and reproducibility, make it an indispensable tool for effective data visualization. These practices ensure that the insights derived from data are communicated in the most efficient and impactful way possible.